J Autom Reasoning (2015) 55:307–372
DOI 10.1007/s10817-015-9327-3

The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations

Part 2—A Survey

Amy P. Felty[1] · Alberto Momigliano[2] · Brigitte Pientka[3]

Received: 24 February 2014 / Accepted: 24 March 2015 / Published online: 5 July 2015

Abstract Over the past three decades, a variety of meta-reasoning systems which support reasoning about higher-order abstract specifications have been designed and developed. In this paper, we survey and compare four meta-reasoning systems, Twelf, Beluga, Abella and Hybrid, using several benchmarks from the open repository ORBI that describes challenge problems for reasoning with higher-order abstract syntax representations. In particular, we investigate how these systems mechanize and support reasoning using a context of assumptions. This highlights commonalities and differences in these systems and is a first step towards translating between them.

Keywords Logical frameworks · Higher-order abstract syntax · Proof assistants · Benchmarks · Context reasoning

✉ Amy P. Felty
afelty@eecs.uottawa.ca

Alberto Momigliano
momigliano@di.unimi.it

Brigitte Pientka
bpientka@cs.mcgill.ca

[1] School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada

[2] Dipartimento di Informatica, Università degli Studi di Milano, Milan, Italy

[3] School of Computer Science, McGill University, Montreal, Canada

1 Introduction

Mechanizing formal systems and proofs about them plays an increasingly important role in designing safe, reliable, and trustworthy software systems in general, and in programming languages in particular; see projects such as LLVM [48] and VTS [3], including CompCert [26] to just name a few. A key question in this endeavor is how to represent the given object language (OL) in a given meta-language (i.e., the language of the logical framework) where the mechanization is carried out. While a wide range of approaches to encoding OLs exists, these approaches vary substantially in how much generic support they provide for common details and bureaucratic infrastructure. Higher-order abstract syntax (HOAS) encodings represent variables in the OL via variables in the meta-language and inherit thereby α-renaming and term-level substitution. Moreover, this encoding technique scales to representing formal systems that use hypothetical and parametrical reasoning by providing generic support for managing hypotheses and the corresponding substitution lemmas. HOAS encodings aim to relieve users from having to build up common infrastructure dealing with variables, assumptions, and substitutions. We consider it the most advanced encoding technique for prototyping formal systems, since it enables us to tackle challenging problems concisely without substantial overhead.

This paper aims to survey the state of the art of several systems supporting HOAS encodings by implementing the ORBI benchmarks (Open challenge problem Repository for systems supporting reasoning with BInders) that we have described in the companion paper [17]. While we have tried to make the present paper self-contained, the kind reader would greatly benefit from reading both papers hand in hand. These benchmark problems are specifically designed to highlight reasoning using a context of assumptions. This allows us to understand this one key aspect, in which all these systems differ.

In this paper, we concentrate on the logical framework Twelf [42], the functional dependently-typed language Beluga [36, 40], the interactive theorem prover Abella [19], and the interactive theorem proving environment Hybrid [16, 32], and show how each of these systems implements the ORBI benchmark problems.

Surveying the state of the art in this area serves as a starting point to understand the commonalities and differences between systems. In particular, we concentrate on the following questions:

- How are contexts represented?
- Which structural properties are enforced and how?
- Do structural properties need to be established individually?
- How do we ensure that all elements in a context are unique?
- How is the substitution property guaranteed?
- How do we relate and reason about different contexts?

We see this as a first step towards formal translation among the different systems. This paper is intended neither to be a tutorial for coding deductive systems with HOAS, for which many excellent ones are readily available, nor as an introduction to the basics of the systems we discuss—in fact, we assume a passing familiarity with them.

We start in Section 2 with a recap of the benchmarks proposed in the companion paper [17]. Then we discuss the formalizations in Twelf and Beluga in Sections 3 and 4, respectively. We present in Section 5, the specification logic used by both Hybrid and Abella, and in Sections 6 and 7 we review the formalization of the benchmarks in these two systems, respectively. Section 8 draws some comparison. Table 1 at the end of that section includes an at-a-glance summary of this comparison, which the expert reader can view in

advance. Finally, we sum up in Section 9. Full details about the challenge problems and their mechanization can be found at https://github.com/pientka/ORBI/.

2 Overview of Benchmarks

Encoding an object logic in a logical framework involves formally representing its syntax, judgments, inference rules, and context structure. In this paper we use the ORBI language proposed in the companion paper [17] to present the OLs that constitute our benchmark problems. The ORBI syntax is currently under development, with the goal of providing a uniform representation of OLs that can be fairly directly translated to systems supporting reasoning with HOAS. In its current version, it supports all of the systems considered in this paper. We give some brief examples in Section 2.1, and refer the reader to Appendix A for all other specifications.

In Section 2.2, we list all of the benchmark theorems from Section 3 of the companion paper using the same numbering as in that paper, so that we may refer to them in subsequent sections as we discuss their proofs in the different systems. We give only a brief explanation of these theorems here.

2.1 Object Logic Specifications

We start with some examples taken from the first benchmark in the companion paper, which considers establishing reflexivity, symmetry, and transitivity of algorithmic equality for the untyped lambda-calculus. The `Syntax`, `Judgments`, and `Rules` sections of an ORBI file adopt the concrete syntax of the Logical Framework (LF) [25]. We recall from Section 4.1 of the companion paper that untyped lambda-terms are introduced with the declaration `tm:type`, followed by the declarations of the constructors `app` with type `tm -> tm -> tm`, and `lam` with type `(tm -> tm) -> tm` in the `Syntax` section. (See Appendix A.1.) The type for `lam` reflects the fact that we represent binders in the object language using binders in the HOAS meta-language.

A predicate representing the algorithmic equality judgment is introduced in the `Judgments` section, followed by object-level inference rules for these judgments in the `Rules` section. For algorithmic equality, we have the predicate `aeq` of type `tm -> tm -> type`, and the following two inference rules, where the ORBI text is a straightforward HOAS encoding of the associated rules.

```
ae_a: aeq M1 N1 -> aeq M2 N2
   -> aeq (app M1 M2) (app N1 N2).
```

$$\frac{\Gamma \vdash \mathsf{aeq}\ M_1\ N_1 \quad \Gamma \vdash \mathsf{aeq}\ M_2\ N_2}{\Gamma \vdash \mathsf{aeq}\ (\mathsf{app}\ M_1\ M_2)\ (\mathsf{app}\ N_1\ N_2)}\ ae_a$$

```
ae_l: ({x:tm} aeq x x -> aeq (M x) (N x))
   -> aeq (lam (\x. M x)) (lam (\x. N x)).
```

$$\frac{\Gamma, x : \mathsf{tm}; \mathsf{aeq}\ x\ x \vdash \mathsf{aeq}\ M\ N}{\Gamma \vdash \mathsf{aeq}\ (\mathsf{lam} x.M)\ (\mathsf{lam} x.N)}\ ae_l$$

Section 2 of the companion paper presents a theory of contexts of assumptions that we use in the benchmarks. There, we present context formation rules defining the structure of contexts and discuss structural properties that contexts should satisfy. We recall here that contexts are made up of *blocks* (also referred to as *declarations*), the semi-colon is used as a separator within blocks, and the comma is used to separate blocks. This structure can be seen in the hypothesis of the ae_l rule above. In other words, we can think of a schema as

originating from the *negative* occurrences of the defined type/predicate (here in the example `tm` and `aeq`) in the given constructors (here `lam` and `ae_l`). In general *context schemas* are used to specify such context structure (see [17] Section 2.2). In Appendix A.1, the following two context schemas are declared in the ORBI `Schemas` section.

```
schema xG:  block (x:tm).
schema xaG: block (x:tm; u:aeq x x).
```

Moreover, *context relations* (see [17] Section 2.4) are defined *inductively* in ORBI in the `Definitions` section. For example, the following definition relates two contexts satisfying the above two schemas.

```
inductive xaR: {G:xG} {H:xaG} prop =
| xa_nil: xaR nil nil
| xa_cons: xaR G H -> xaR (G, block (x:tm)) (H, block (x:tm; u:aeq x x)).
```

In the next subsection we will write (`xG` Φ_x) and (`xaG` Φ_{xa}) to mean that the contexts Φ_x and Φ_{xa} satisfy the above two schemas, respectively, and (`xaR` Φ_x Φ_{xa}) to say that the pair of contexts satisfies the above relation. By convention the suffix `G` will be used for context schemas and `R` for n-ary context relations. To look up an assumption in a context, we simply write $A \in \Gamma$, meaning that there is some block D in context Γ such that $A \in D$. We will also overload the notation, writing $D \in \Gamma$ to indicate that a context contains an entire block.

Contexts schemas can have *alternatives*. For example, one of our benchmark problems extends the notion of algorithmic equality to the polymorphically-typed lambda-calculus. In addition to `app` and `lam`, this OL includes the constants `tapp` of type `tm -> tp -> tm` and `tlam` of type `(tp -> tm) -> tm` for type application and type abstraction, respectively, where `tp` is the constant introduced to represent object-level types. (See Appendix A.2.) The following two context schemas from that benchmark include an alternative for types, and thus contexts satisfying these schemas may contain blocks of both forms.

```
schema axG: block (a:tp) + block (x:tm).
schema aeqG: block (a:tp; u:atp a a) + block (x:tm; v:aeq x x).
```

In the formal proofs in the rest of this paper, we will refer to the structural properties of contexts in Section 2.3 of the companion paper. We do not repeat them here, but just note that we distinguish between weakening, strengthening, and exchange within a single *d*eclaration and over a *c*ontext as a whole. The former are called *d-wk*, *d-str*, and *d-exc*, while the latter are called *c-wk*, *c-str*, and *c-exc*.

2.2 A Recap of the Benchmark Theorems

The benchmarks in the companion paper are structured around different shapes and properties of contexts. To help the reader we repeat here the classification after some initial considerations, but we refer to the companion paper for its motivations.

All of these benchmarks include both G and R versions of the theorem statements. These two approaches are discussed in detail in the companion paper. Here, we simply note that the G approach uses a common context for all judgments in the statement of a theorem, while the R version may use different contexts; in other words, an R-version proof *explicitly*

relates two or more contexts. The reader is again referred to Appendix A for all definitions of context schemas and context relations.

Some special care is required w.r.t. inference rules for well-formed terms and types, as presented in Section 2.4 of the companion paper. Such inference rules can be inferred from the declarations in the `Syntax` section of an ORBI specification. We do not present them explicitly, but just note here that we name them by prefixing `is_` to the judgment name. In the statements below, we have `is_tm` and `is_tp`. In general, these predicates are used in theorem statements when the proof involves induction over the well-formedness inference rules (e.g., Theorems 7 and 8). We also implicitly assume that if `x:tm` $\in \Phi$, then `[`$\Phi \vdash$ `is_tm x]`, and similarly for `tp` and `is_tp`.

Basic Linear Context Extensions The first benchmark (algorithmic equality for the untyped lambda-calculus) includes Lemmas and Theorems 6–10. See Appendix A.1 for the full listing of syntax, rules, schemas and context relations involved in this benchmark and [17, Section 3.1] for the informal proofs of the lemmas and theorems.

Lemma 6 (**Context Membership**)
(`xaR`$\Phi_x\,\Phi_{xa}$) *implies that* (`block(x:tm)`)$\in\Phi_x$ *iff* (`block(x:tm;u:aeq x x)`)$\in\Phi_{xa}$.

Theorem 7 (**Admissibility of Reflexivity, R Version**)
Assume (`xaR` Φ_x Φ_{xa}). *If* `[`$\Phi_x \vdash$ `is_tm M]` *then* `[`$\Phi_{xa} \vdash$ `aeq M M]`.

Theorem 8 (**Admissibility of Reflexivity, G Version**)
Assume (`xaG` Φ_{xa}). *If* `[`$\Phi_{xa} \vdash$ `is_tm M]` *then* `[`$\Phi_{xa} \vdash$ `aeq M M]`.

Lemma 9 (**Context Inversion**)
Assume (`xaG` Φ_{xa}). *If* `u:aeq M N` $\in \Phi_{xa}$, *then* `M = N`.

Theorem 10 (**Admissibility of Symmetry and Transitivity**)
Assume (`xaG` Φ_{xa}).

1. *If* `[`$\Phi_{xa} \vdash$ `aeq M N]` *then* `[`$\Phi_{xa} \vdash$ `aeq N M]`
2. *If* `[`$\Phi_{xa} \vdash$ `aeq M L]` *and* `[`$\Phi_{xa} \vdash$ `aeq L N]` *then* `[`$\Phi_{xa} \vdash$ `aeq M N]`.

Linear Context Extensions with Alternative Declarations We extend here (Lemmas and Theorems 11–17) reflexivity, symmetry, and transitivity of algorithmic equality to the polymorphic lambda-calculus. See Appendix A.2 and [17, Section 3.2].

Theorem 11 (**Admissibility of Reflexivity for Types, G Version**)
Assume (`atpG` Φ_{atp}). *If* `[`$\Phi_{atp} \vdash$ `is_tp A]` *then* `[`$\Phi_{atp} \vdash$ `atp A A]`.

Lemma 12 (**G-Promotion for Type Reflexivity**)
Assume (`aeqG` Φ_{aeq}). *If* `[`$\Phi_{aeq} \vdash$ `is_tp A]` *then* `[`$\Phi_{aeq} \vdash$ `atp A A]`.

Theorem 13 (**Admissibility of Reflexivity for Terms, G Version**)
Assume (`aeqG` Φ_{aeq}). *If* `[`$\Phi_{aeq} \vdash$ `is_tm M]` *then* `[`$\Phi_{aeq} \vdash$ `aeq M M]`.

Theorem 14 (**Admissibility of Reflexivity for Types, R Version**)
Assume (`atpR` Φ_α Φ_{atp}). *If* `[`$\Phi_\alpha \vdash$ `is_tp A]` *then* `[`$\Phi_{atp} \vdash$ `atp A A]`.

Lemma 15 (**Relational Strengthening**)
Assume (aeqR $\Phi_{\alpha x}$ Φ_{aeq}). *Then there exist contexts* Φ_α *and* Φ_{atp} *such that* (alphxR $\Phi_{\alpha x}$ Φ_α), (aeqatpR Φ_{aeq} Φ_{atp}), *and* (atpR Φ_α Φ_{atp}).

Lemma 16 (**R-Promotion for Type Reflexivity**)
Assume (aeqR $\Phi_{\alpha x}$ Φ_{aeq}). *If* [$\Phi_{\alpha x} \vdash$ is_tp A] *then* [$\Phi_{aeq} \vdash$ atp A A].

Theorem 17 (**Admissibility of Reflexivity for Terms, R Version**)
Assume (aeqR $\Phi_{\alpha x}$ Φ_{aeq}). *If* [$\Phi_{\alpha x} \vdash$ is_tm M] *then* [$\Phi_{aeq} \vdash$ aeq M M].

Non-Linear Context Extensions Here (Lemmas and Theorems 18–22), we go back to the untyped lambda-calculus, introduce declarative equality, and prove the completeness of algorithmic equality with respect to it. See Appendix A.1 and [17, Section 3.3].

Lemma 18 (**G-Promotion for Reflexivity, Symmetry, and Transitivity**)
Assume (daG Φ_{da}).

1. *If* [$\Phi_{da} \vdash$ is_tm M] *then* [$\Phi_{da} \vdash$ aeq M M].
2. *If* [$\Phi_{da} \vdash$ aeq M N] *then* [$\Phi_{da} \vdash$ aeq N M].
3. *If* [$\Phi_{da} \vdash$ aeq M L] *and* [$\Phi_{da} \vdash$ aeq L N] *then* [$\Phi_{da} \vdash$ aeq M N].

Theorem 19 (**Completeness, G Version**)
Assume (daG Φ_{da}). *If* [$\Phi_{da} \vdash$ deq M N] *then* [$\Phi_{da} \vdash$ aeq M N].

Lemma 20 (**Relational Strengthening**)
Assume (daR Φ_{xa} Φ_{xd}). *Then there exists a context* Φ_x *such that* (xaR Φ_x Φ_{xa}).

Lemma 21 (**R-Promotion for Reflexivity**)
Assume (daR Φ_{xa} Φ_{xd}). *If* [$\Phi_{xd} \vdash$ is_tm M] *then* [$\Phi_{xa} \vdash$ aeq M M].

Theorem 22 (**Completeness, R Version**)
Assume (daR Φ_{xa} Φ_{xd}). *If* [$\Phi_{xd} \vdash$ deq M N] *then* [$\Phi_{xa} \vdash$ aeq M N].

Order The next benchmark considers order in a context using the same OL (the untyped lambda-calculus). See [17, Section 3.4].

Theorem 23 (**Pairwise Substitution**)
Assume (xaG Φ_{xa}). *If* [Φ_{xa}, block (x:tm; u:aeq x x) $\vdash$ aeq (M1 x) (M2 x)] *and* [$\Phi_{xa} \vdash$ aeq N1 N2], *then* [$\Phi_{xa} \vdash$ aeq (M1 N1) (M2 N2)].

Uniqueness The next benchmark problem is type uniqueness of the simply-typed lambda-calculus. See Appendix A.3 and [17, Section 3.5].

Theorem 24 (**Type Uniqueness**)
Assume (xtG Φ_t). *If* [$\Phi_t \vdash$ oft M A] *and* [$\Phi_t \vdash$ oft M B], *then* A = B.

Substitution The final benchmark addresses the interaction of the substitution property with context reasoning. The OL is again the simply-typed lambda calculus, and judgments

include a parallel reduction relation, which evaluates terms under lambda-abstractions. See Appendix A.4 and [17, Section 3.6].

Lemma 25 *Assume* (`xtG` Φ_t). *If* `[`Φ_t`, block (x:tm; v:oft x A)` $\vdash$ `oft (M x) B]` *and* `[`$\Phi_t \vdash$ `oft N A]`, *then* `[`$\Phi_t \vdash$ `oft (M N) B]`.

Theorem 26 (**Type Preservation for Parallel Reduction, R Version**)
Assume (`xrtR` Φ_r Φ_t). *If* `[`$\Phi_r \vdash$ `pr M N]` *and* `[`$\Phi_t \vdash$ `oft M A]`, *then* `[`$\Phi_t \vdash$ `oft N A]`.

Theorem 27 (**Type Preservation for Parallel Reduction, G Version**)
Assume (`xrtG` Γ). *If* `[`$\Gamma \vdash$ `pr M N]` *and* `[`$\Gamma \vdash$ `oft M A]`, *then* `[`$\Gamma \vdash$ `oft N A]`.

3 Mechanization in Twelf

Twelf supports specifying formal systems in the logical framework LF [25]. LF is a very compact dependent type theory supporting atomic types a that can be indexed by terms M_i and (dependent) function types $\Pi x{:}A.B$. Following recent practice, we only characterize terms in normal form, since these are the only ones that are meaningful.

$$\begin{array}{lll} \text{Kind} & K & ::= \mathsf{type} \mid \Pi x{:}A.K \\ \text{Types} & A, B & ::= a\ M_1 \ldots M_n \mid \Pi x{:}A.B \\ \text{Terms} & M, N & ::= x\ M_1 \ldots M_n \mid c\ M_1 \ldots M_n \mid \lambda x.M \end{array}$$

Logically, we can think of $a\ M_1 \ldots M_n$ as a predicate. Implications correspond to the non-dependent function space $\Pi x{:}A.B$ where x does not occur in B. This is commonly abbreviated as $A \to B$. Universal quantification corresponds to the dependent function space $\Pi x{:}A.B$.

The grammar of LF does not separate between the types for terms and those for predicates. Rather, this distinction happens naturally on the level of kinds that classify types. A type family of level 0 has kind `type`. The type `tm` from the previous section is an example. A type family that depends on objects of level 0 such as `aeq` is a type of level 1, etc. Note the uniform treatment of binding structures. Not only are object-level binders on the level 0 modeled via a function space and hence naturally support object-level substitution via function application, but hypothetical and parametric derivations (i.e., objects of level 1) are characterized by a (dependent) function. Hence, substituting into a given derivation also boils down to function application. Since non-dependent functions are a special case of dependent functions, both the parametric and hypothetical substitution property collapses to the same substitution operation. As a consequence this methodology easily scales to encoding and reasoning about *intrinsically* typed systems instead of stating the well-typedness relation separately. For a general introduction on how to encode object languages in LF, we recommend [34].

3.1 Basic Linear Context Extensions (G Version)

In Twelf proofs are implemented as a *relation* between derivations within the logical framework LF. For a summary of the specification of algorithmic equality, referred to below as `aeq M N`, we refer the kind reader to Appendix A.1. We begin by stating reflexivity of algorithmic equality (Theorem 8) as a relation `ref` between a term `M` and the algorithmic equality judgment `aeq M M`. It reads as: For all `M:tm` there exists a derivation `aeq M M`.

We implement the proof by analyzing and inducting on `M` directly, since there is no essential distinction between terms (type family of level 0) and predicates (type families of level 1). As a consequence, we do not have to encode a predicate `is_tm M` reifying the definition of terms. We note that `ref` depends on `tm` (type family of level 0) and `aeq` (type family of level 1) and hence `reflG` is a type family of level 2.

```
reflG: ΠM:tm.aeq M M → type.

r_a: reflG M2 D2 → reflG M1 D1
  → reflG (app M1 M2) (ae_a D1 D2).

r_l: (Πx:tm.Πu:aeq x x.Πr_x: reflG x u. reflG (M x) (D x u))
      → reflG (lam λx. M x) (ae_l λx.λu. D x u)
```

The reflexivity proof is then implemented using two constants `r_a` and `r_l`. We can read `r_a` as follows: Given that `reflG M2 D2` (IH on `M2`) and `reflG M1 D1` (IH on `M1`), we can obtain a proof `reflG (app M1 M2) (ae_a D1 D2)`. The definition of `ae_a` and `ae_l` are given in Appendix A.1.

The case for lambda-terms is slightly more tricky. In Twelf, not only specifications but also proofs are represented in the implicit-context style. As a consequence the variable cases in a proof are treated for each variable that is introduced as we traverse a binder. Hence our recursive call (IH) will take place under the assumption that `x:tm`. Moreover, we cannot simply appeal to `reflG (M x) (D x u)`. First, we must know that `u:aeq x x`. Second, we must have that the reflexivity relation holds for the variable `x`. The base case, which in the on paper proof is handled separately, is folded into the case for lambda-abstraction. We can therefore read `r_l` as: Assuming that for all `x:tm` and `u:aeq x x`, `reflG x u` implies `reflG (M x) (D x u)`, we can prove that `reflG (lam λx.M x) (ae_l λx.λu.D x u)`.

LF type checking guarantees that all derivations that are manipulated and constructed are meaningful. The context of assumptions is ambient and implicit. However, for the relation `reflG` to be a proof, we also must establish that it implements a *total function*; in particular, we must guarantee that all the necessary base cases are taken care off. This cannot be achieved within the logical framework LF. Instead, Twelf relies on *external checkers*. There are three distinct properties we need to establish about the relation:

1. Well-modedness. The mode checker [41] establishes that the given relation can be viewed as a non-deterministic function. In the given example, we specify that `M` is an input and `D` is the output of the relation using a mode declaration

   ```
   %mode reflG +M -D.
   ```

   Mode checking verifies that whenever the inputs are given the specified output can be computed; it does not, however, guarantee that there is a unique output value.
2. World satisfaction. The world checker verifies that the implemented relation introduces dynamic assumptions according to the predefined schema.

   ```
   %block  xaG_r : block  {x:tm}{u:aeq x x}{r_x:reflG x u}.
   %worlds  (xaG_r) (reflG M D).
   ```

   Thus, a context schema is formalized by a **`%block`** declaration and schema satisfaction by the **`%worlds`** check that verifies that indeed the reflexivity proof only introduces the specified dynamic assumptions. Twelf only permits the generalized contexts approach

(G version), since assumptions are ambient. Note that the schema defined differs from the schema definition `xaG` in the ORBI file, since in addition to `x:tm` and `u:aeq x x` the declaration also contains `reflG x u`. The defined schema of the eligible contexts is more specific and more restrictive than our definition in the companion paper. This is tied to the fact that we state proofs as relations using contexts implicitly. World checking is essential in Twelf to guarantee that all base cases are handled.

3. Totality. This checks two properties: termination and coverage. The termination checker [35] verifies that the given relation is terminating according to a specified ordering, while the coverage checker verifies that a case exists for all possible inputs.

```
%total   M (reflG M D).
```

We state that the relation `reflG` terminates because it is defined recursively on `D` and all appeals to the IH are structurally smaller.

The coverage checker [44] verifies that a given relation is covering, i.e., a case for all possible inputs is provided and splitting on an output argument does not neglect and drop possible cases.

Twelf's meta-language to establish that a given relation constitutes a total function is restricted to the language $\mathcal{M}^2$ [43], a fragment of first-order logic that only admits ∀-∃ formulas and does not allow arbitrary nesting of universal and existential quantification.

3.2 Linear Context Extensions with Alternative Declarations (G Version)

Extending the algorithmic equality case study to the polymorphic lambda-calculus is straightforward. For the specification of types, terms, algorithmic and declarative equality, we add the additional cases for handling type abstractions and applications (see Appendix A.2).

Since types can occur in terms, we need to establish additional lemmas about the admissibility of reflexivity, symmetry and transitivity on the level of types. We only give the encoding of the reflexivity statement: For all `T:tp`, there exists a derivation `atp T T`, together with the meta-theoretic directives for mode, worlds, and totality. We note again that we implement the proof by recursion over `T:tp`.

```
reflTpG: Π T:tp.atp T T → type .
%mode reflTpG +T -D.
%block  atpG_r : block  {a:tp}{u:atp a a}{r_a:reflTpG a u}.
%worlds  (atpG_r) (reflTpG T D).
%total   T (reflTpG T D).
```

Next, we extend the reflexivity proof `reflG` for terms with the additional cases for type abstraction and type application. We call the extended proof `reflTmG` to emphasize it considers terms only. There are two issues that arise: First, since the case for lambda-abstraction introduces assumptions about term variables and the equality of term variables, while the case for type abstraction introduces type variables and equality of type variables, we need to specify *alternating* assumptions in a world declaration. This alternation of blocks is described by the following world declaration `(xaG_r | atpG_r)`.

```
%block   xaG_r : block  {x:tm}{u:aeq x x}{r_x:reflTmG x u}.
%worlds  (xaG_r | atpG_r) (reflTmG T D).
```

The admissibility of reflexivity is then established in the world (aka context) containing both blocks `xaG_r` and blocks `atpG_r`. The world checker then verifies that our context contains blocks of declarations consisting of `x:tm; u:aeq x x; r_x:reflTmG x u` *or* `a:tp; u:atp a a; r_a:reflTpG a u`.

Second, the reflexivity proof `reflTmG` for terms relies on the reflexivity proof `reflTpG` for types. This is justified by the fact that the world for `reflTpG` is a strict subset of the world specified for `reflTmG`, where the additional assumptions in block `xaG_r` are irrelevant to the reflexivity proof `reflTpG` for types. This is facilitated by Twelf's *subordination* analysis, which maintains a dependency graph of all the type families. If a type family a is not a subordinate to a type family b, then elements of type a cannot be used to construct elements of type b; hence declarations formed by the type family a can safely be added to (i.e., weakening) and removed from (i.e., strengthening) the context. Therefore, one might say subordination enables safe instances of context weakening and context strengthening (see [24]).

A similar issue arises in the transitivity and symmetry property for terms, which relies on the corresponding property for types.

3.3 Non-Linear Context Extensions (G Version)

We now inspect the implementation of the completeness proof of Theorem 19 given in Fig. 1, which only considers equality on terms instead of polymorphic terms. We again define a type family `ceqG` describing the relation between `deq M N` and `aeq M N` (see Appendix A.1 for their specifications). We omit quantifying over `M` and `N` explicitly leaving them free and let Twelf infer the type of these variables. Furthermore, we sometimes write _ at a given argument position letting Twelf's reconstruction engine infer the concrete instantiation (see for example the case for `c_r`).

When considering each case in the proof, we need to pay attention to which implicit assumptions are needed. We consider each case for `deq T S` separately:

If `de_r:deq M M` was used, then we appeal to the reflexivity lemma. We note that the reflexivity lemma is proven in a context containing `xaG_r`, containing blocks of assumptions `x:tm`, `u:aeq x x` and `r_x:reflG x u`. Therefore, we want the context for `ceqG` to contain at least those assumptions described in `xaG_r`. We say here "at least", since the context in which we prove `ceqG` may contain more assumptions than needed by

```
ceqG: deq M N → aeq M N → type .
c_r: reflG _ E → ceqG de_r E.

c_t: ceqG D1 E1 → ceqG D2 E2 → transG E1 E2 E → ceqG (de_t D1 D2) E.

c_s: ceqG D E1  →  sym E1 E → ceqG (de_s D) E.

c_l:(Πx:tm.Πu:aeq x x.Πr_x:reflG x u.Πt_x:transG u u u.Πs_x:symG u u.
     Πv:deq x x. ceqG v u → ceqG (D x v) (E x u))
     → ceqG (de_l λx.λv.D x v) (ae_l λx.λu.E x u).

c_a: ceqG D1 E1 → ceqG D2 E2 → ceqG (de_a D1 D2) (ae_a E1 E2).

%mode ceqG +D -E.
%block  daG_rtsc : block  {x:tm}{u:aeq x x}
                          {r_x:ref x u}{t_x:tr u u u}{s_x:sym u u}
                          {v:deq x x}
                          {c_x:ceqG v u}.
%worlds  (daG_rtsc) (ceqG D E).
%total   D (ceqG D E).
```

Fig. 1 Completeness of algorithmic equality in Twelf

reflG, provided that such assumptions do not contribute to the proof of reflG. This is again enforced in Twelf by subordination analysis. If de_t was used, we have two sub-derivations: D1:deq M L and D2:deq L N. By induction, we obtain derivations E1:aeq M L and E2:aeq L N. To finish the proof, we appeal to the transitivity lemma. We note that the transitivity lemma was proven in a context containing the block x:tm, u:aeq x x, and t_x:transG u u u. The case for symmetry using de_s is similar. The implicit context for the completeness proof must therefore contain the union of the assumptions needed for applying the reflexivity, transitivity, and symmetry lemmas.

The case for application is straightforward. Finally, we consider the case for lambda-abstractions, where we have used de_l, declarative equality for lambda-terms, and the context is extended. Intuitively, we want to introduce a variable x together with the assumption v:deq x x and u:aeq x x, and simply appeal to the IH by making the following recursive call: ceqG (D x v) (E x u). However, since all the information pertaining to variables is folded into the binder-case, we also need to add the following assumptions: r_x:reflG x u, t_x:transG u u u, s_x:symG u u and ceqG v u; in other words, we must assume that the reflexivity, symmetry and transitivity lemmas hold for the variable x in addition to assuming that indeed there is a proof ceqG v u that guarantees that whenever we have deq x x we must have a proof for aeq x x.

To verify that the implemented relation constitutes a proof, we again check well modedness, world satisfaction, and totality. World checking will make sure that we can safely refer to the transitivity and reflexivity lemmas, which both are proven in a smaller context. To ensure that this check is reasonably efficient, Twelf verifies that all blocks occurring in the context of the lemma occur in the context of the main theorem; in addition, the type families in the lemma (i.e., callee) must be subordinate to the additional type families in the theorem (i.e., caller). This guarantees that the additional type families in the theorem are irrelevant to the lemma.

This notion of *context subsumption* is a sufficient condition for weakening and strengthening the contexts associated with a given type family and obviates in many cases the need to prove weakening and strengthening lemmas. However, it can also be brittle, since the order of assumptions matters.

In Twelf-1.7.1 we can weaken a block by inserting additional assumptions that are irrelevant to the callee, but we cannot reorder assumptions. This additional flexibility alleviates some of the issues that arise due to the fact that base cases must be included in the block declarations.

For a more detailed explanation regarding mechanizing proofs in the Twelf system and context subsumption, we refer the reader to [24].

3.4 Order (G Version)

To encode the pairwise substitution lemma (Theorem 23) we cannot rely simply on LF function application. The code is reported in Fig. 2; specification of algorithmic equality, i.e., aeq M N is given in Appendix A.1. Exchange is handled straightforwardly. However, an interesting issue arises in the base cases. We have one base case s_x, where x is the variable we want to replace and u:aeq x x describes the derivation that x is equal to itself. In this case we simply return the input derivation F:aeq N1 N2. The second base case arises when traversing a lambda-abstraction. Here we encounter a variable y together with the assumption ae_y:aeq y y and we want to return the latter no matter what we substitute for x. We hence assume ΠN1:tm.ΠN2:tm.ΠF:aeq N1 N2.subst(λx.λae_x.ae_y) F ae_y that can be

```
subst: (Πx:tm.aeq x x → aeq (M1 x) (M2 x)) → aeq N1 N2
      → aeq (M1 N1) (M2 N2) → type .

s_x: subst (λxλu. u) F F.

s_l: (Πy:tm.Πae_y:aeq y y.
      (ΠN1:tm.ΠN2:tm.ΠF:aeq N1 N2.subst (λx.λae_x. ae_y) F ae_y) →
       subst (λx.λu. D x u y v) F (E y v))
       → subst (λx.λu.ae_l λy.λv. D x u y v) F (λx.λu. ae_l E x u).

s_a: subst (λx.λu. D1 x u) F E1 → subst (λx.λu. D2 x u) F E2
      → subst (λx.λu. ae_a (D1 x u) (D2 x u)) F (ae_a E1 E2).

%mode subst +D +F -E.
%block  xaG_s:block {y:tm}{ae_y:aeq y y}
        {s_y:ΠN1:tm.ΠN2:tm.ΠF:aeq N1 N2. subst (λx.λae_x.ae_y) F ae_y}.
%worlds  (xaG_s) (subst D F E).
%total   D (subst D F E).
```

Fig. 2 Pairwise substitution lemma in Twelf

read as: For all terms `N1,N2` and derivation `F:aeq N1 N2`, the result of substituting `F` for `aeq x x` is simply `ae_y`. This is an example of a higher-order assumption that Twelf supports.

3.5 Uniqueness (G Version)

We consider next the implementation of the proof of type uniqueness for simply typed lambda-terms. The specification of terms together with their typing rules can be found in Appendix A.3. To express the relation between a term `M` and its corresponding type `T`, we write `oft M T`. As Twelf has no built-in equality, we must define it. There is a choice to define equality simply using reflexivity, `e_ref:eq T T`, or recursively based on the structure of the type. This choice surprisingly matters in Twelf. We sketch the first approach in Fig. 3, by defining a relation `unique`. The implementation of the proof follows earlier ideas with one exception. We cannot simply refer to `e_ref`. The reason has to do with type reconstruction, which will consider the inputs and outputs of a given relation. Let's for example consider the case (denoted by `u_l`) where we have used the typing rule for

```
% Inversion lemmas about function types
arr_inv1:  eq (arr T1 T2) (arr T1' T2') → eq T2 T2' → type .
arr_inv1_ref: arr_inv1 e_ref e_ref.
...
arr_inv2: ΠT:tp. eq T2 T2' → eq (arr T T2) (arr T T2') → type.
arr_inv2_ref: arr_inv2 T  e_ref e_ref.
%mode arr_inv2 +T +E1 -E2.
%worlds  () (arr_inv2 T E1 E2).
%total   E1 (arr_inv2 _ E1 _).

% Type uniqueness proof
unique: oftM T → oftM T' → eq T T' → type.
u_a: arr_inv1 E E' → unique D1 F1 E
    → unique (of_a D1 D2) (of_a F1 F2) E'.

u_l: arr_inv2 T1 E E'
   → (Πx:tm.Πu:oft x T1.unique u u e_ref → unique (D x u) (F x u) E)
   → unique (of_l λx.λu. D x u) (of_l λx.λu. F x u) E'.

%mode unique +D +F -E.
%block  xtG_u: some {T:tp} block  {x:tm}{d:oft x T}{u:unique d d e_ref}.
%worlds  (xtG_u) (unique D F E).
%total   D (unique D _ _).
```

Fig. 3 Type uniqueness in Twelf

lambda-terms. When we write `e_ref` in the output position in clause `u_l`, we will restrict what `T` and `T'` can be—effectively, we are setting them to be identical. This will cause coverage to fail (correctly), because there is in fact no case where the types are different. Relations unlike functions cannot always capture the flow and refinement of information that happen in the proof. One solution is to prove some inversion lemmas for function types and then appeal to them, but avoid unduly restricting arguments in the main theorem. We again rely on subordination to justify calling the lemmas `arr_inv1` and `arr_inv2`, which are defined in the empty context (world) within the context `xtG_u`. Finally, we remark that to avoid adding for each individual variable the base case `unique u u e_ref`, we could omit it and add instead a generic base case `u_b:unique D D e_ref`. This would allow us to define a tighter context only consisting of `x:tm` and `d:oftx T`.

A different approach is to define equality on the structure of the type as a judgment `atp T T` and then prove that reflexivity is admissible without further ado, avoiding the previous coverage problems, but reasoning structurally about equality.

3.6 Substitution

Twelf models hypothetical and parametric derivations as LF functions. Therefore, substitution can be obtained for free relying on LF function application. This is illustrated in the type preservation proof for parallel reductions. Parallel reductions, described by `pr M N`, together with typing rules, described by `oftN A`, are specified in Appendix A.4. We define Theorem 27 that states that if `pr M N` and `oftM A`, then `oftN A`, and its corresponding proof in Fig. 4.

We want to draw particular attention to the case `tp_b` that considers the case for beta-reduction. The remaining cases can be found in the online files. We consider the reduction of `(lam λx.M x) N` (rule `pr_b` in Appendix A.4). `R1` denotes a parallel reduction sequence for `pr (M x) (M' x)`, given that `pr x x`. The reduction sequence for `pr N N'` is described by `R2`. By the IH on `R1`, we obtain a derivation `F1:Πx:tm.oftx A →oft(M x) B`. Moreover, by the IH on `R2`, we obtain a derivation `F2:oftN A`. We now want to substitute `N` for `x` in `F1` and subsequently `F2` for the assumption `oftN A` in `F1`. Since `F1` denotes a function, we simply use function application to model substitution writing `(F1 N F2)`.

4 Mechanization in Beluga

Beluga is a proof and programming environment [36, 39, 40] that also supports the specification of formal systems in the logical framework LF. On top of LF Beluga provides a

```
tps: pr M N → oftM A → oftN A → type.
tp_b: tps (pr_b R1 R2) (of_app (of_lam D1) D2) (F1 N F2)
      ← (Πx:tm.Πof_v:oft x A.Πpr_v:pr x x. tps pr_v of_v of_v
          → tps (R1 x pr_v) (D1 x of_v) (F1 x of_v))
      ← tps R2 D2 F2.

%mode tps +R +D -F.
%block  xrtG_tpv: some {t:tp} block  {x:tm}{of_v:oft x t}{pr_v:pr x x}
                                     {tp_v:tps pr_v of_v of_v}.
%worlds  (xrtG_tpv) (tps R D F).
%total   R (tps R D F).
```

Fig. 4 Type preservation in Twelf (only the case for β-reduction)

dependently typed functional language that allows programmers to manipulate and analyze contexts and contextual LF objects, i.e., a LF object M in a context of assumptions Ψ [33], which we write as [Ψ ⊢ M]. The dependency on a context is also captured on the level of types via the contextual type [Ψ ⊢ A]. We say a contextual object [Ψ ⊢ M] has type [Ψ ⊢ A] if M has type A in the context Ψ. This allows us to concisely represent and manipulate hypothetical and parametric derivations, i.e., a goal formula (object) together with the assumptions that we are allowed to use to establish the goal.

Beluga's type language takes contextual types as base types; it also supports product types (T_1 * T_2) and function types ($T_1 \rightarrow T_2$), and we can universally quantify over contextual objects and contexts ({X:U}T where U is either a contextual type [Ψ ⊢ A] or describes a context schema, i.e., the type of a context). Logically, Beluga's type language corresponds to first-order logic where we quantify over contextual objects and contexts and take contextual objects as our basic predicates.

Beluga's computation language supports tuples (introduction for product types), nameless functions (fn x⇒e) (introduction for function types), abstraction over contextual objects and contexts ($\lambda^{\square}$ ⇒e) (introduction for universal types), recursion (rec f⇒e), and case-analysis over contextual objects and contexts. Unlike Twelf's meta-logic, which prevents nesting of quantifiers and implications, Beluga's computation language imposes no such restrictions and supports higher-order functions. In this section, we will concentrate on showing how to implement inductive proofs from our benchmarks using the generalized approach (G version); we also briefly discuss how to encode proofs in the relational approach.

Inductive proofs in Beluga are represented as recursive functions about contextual LF objects using pattern matching. Each case of the proof corresponds to one branch in the function. Recently, Cave and Pientka have shown how to extend Beluga with support for inductive definitions (i.e., recursive datatypes) [8] and first-class substitutions [9] leading to Beluga$^{\mu}$. This allows us to define predicates about contexts and contextual objects, relate contexts via substitutions, and more importantly prove properties about them inductively.

4.1 Basic Linear Context Extensions (G Version)

We consider first the proof that reflexivity is admissible for algorithmic equality (Theorem 8). For a summary of the specification of algorithmic equality, written as aeq M N, we refer the kind reader to Appendix A.1. Just as types classify expressions, contexts are classified by context schemas and the definition of schemas in the ORBI specification directly translates to Beluga.[1]

```
schema xaG = block (x:tm,u:aeq x x);
```

The schema states that our context consists of a block of assumptions, containing x:tm and aeq x x. More formally, the block-construct introduces a Σ-type grouping the two declarations together. Schemas are closely related to the world declarations in Twelf. However, there is a crucial difference. In Twelf a world declaration also contains the base cases in a proof; for example, in the reflexivity proof, the world declaration also contained ref x u, a proof that the statement holds for each x we encounter. In Beluga it is sufficient to keep track of x:tm and u:aeq x x, as we are abstracting over the context and will

[1] Beluga's syntax for schemas differs slightly from ORBI's syntax.

```
rec reflG: {γ:xaG}{M:[γ ⊢ tm]} [γ ⊢ aeq (M ..) (M ..)] =
λ□γ ⇒ λ□M ⇒ case [γ ⊢ M ..] of
| [γ ⊢ #p.1 ..] ⇒ [γ ⊢ #p.2 ..]                                  % Variable

| [γ ⊢ lam λx. M .. x] ⇒                                          % Lambda
  let  [γ, b:block(y:tm,ae_v:aeq y y) ⊢ D .. b.1 b.2]=
         reflG [γ, b:block(y:tm,ae_v:aeq y y)] [γ, b ⊢ M .. b.1]
  in  [γ ⊢ ae_l λx. λw. (D .. x w)]

| [γ ⊢ app (M1 ..) (M2 ..)] ⇒                                    % Application
  let  [γ ⊢ D1 ..] = reflG [γ] [γ ⊢ M1 .. ] in
  let  [γ ⊢ D2 ..] = reflG [γ] [γ ⊢ M2 .. ] in
    [γ ⊢ ae_a (D1 ..) (D2 ..)];
```

Fig. 5 Reflexivity of algorithmic equality in Beluga

be able to write a generic variable case covering the scenario where we consider a variable from the context. Moreover, schema checking in Beluga, unlike world checking in Twelf, does not check that a given type family satisfies the declared schema; it merely ensures that contexts that are constructed and passed as arguments in computations satisfy the declared schema.

The reflexivity theorem (Theorem 8) can be implemented as a recursive function called `reflG` of type: $\{\gamma:\texttt{xaG}\}\{\texttt{M}:[\gamma \vdash \texttt{tm}]\}[\gamma \vdash \texttt{aeq (M..) (M..)}]$ (see Fig. 5). The type can be read as: For all contexts γ that have schema `xaG`, for all terms `M` that are meaningful in γ, we have $[\gamma \vdash \texttt{aeq (M..) (M..)}]$. We explicitly quantify over contexts and contextual objects in computation-level types using curly braces and the corresponding abstraction on the level of expressions is written as $\lambda^{\square}\gamma \Rightarrow \lambda^{\square}\texttt{M} \Rightarrow \texttt{e}$. As in Twelf, we do not need to reify the definition of terms with a predicate `is_tm` to reason inductively about terms, as Beluga supports directly induction on universally quantified objects.

Central is the idea of a contextual type. For example, `M` has type $[\gamma \vdash \texttt{tm}]$, which describes an object `M` that has type `tm` in the context γ. Hence, `M` is a term that may refer to variables in the context γ. When we use `M` it is associated with a substitution that maps all the variables in γ to the correct target context. In the example, we use `M` within the contextual type $[\gamma \vdash \texttt{aeq (M..) (M..)}]$. Hence, `M` is declared in the context γ and because it is also used in the context γ, it is associated with the identity substitution, which is written as .. in our concrete syntax. Intuitively, it means that `M` can depend on all the variables that occur in the context described by γ. The derivation $\gamma \vdash \mathsf{aeq}\ M\ M$ is directly captured by the contextual type $[\gamma \vdash \texttt{aeq (M..) (M..)}]$. While the informal on paper definitions often leave implicit the question in what context a term is well-formed, Beluga's concrete syntax forces the programmer to be more precise.

In the proof for `reflG` we begin by introducing γ and `M` followed by a case analysis on $[\gamma \vdash \texttt{M..}]$ using pattern matching. There are three possible cases for `M`:

1. Variable case. If `M` is a variable from γ, we write $[\gamma \vdash \texttt{\#p.1..}]$ where `#p` denotes a parameter variable declared in the context γ. Operationally, `#p` can be instantiated with any bound variable from the context γ. Since the context γ has schema `xaG`, it contains blocks `x:term,ae_v:aeq x x`. The first projection of it allows us to extract the term component, while the second projection of it denotes the proof of `aeq x x`.
2. Lambda case. If `M` is a lambda-term, then we extend the context and appeal to the induction hypothesis by making a recursive call. Beluga supports declaration weakening (see the companion paper [17]); this justifies using the term `M`, which has type $[\gamma,\texttt{x:tm} \vdash \texttt{tm}]$, in the context $[\gamma,\texttt{b:}\mathbf{block}\texttt{(x:term,ae\_v:aeq x x)}]$. To move from domain $\gamma,\texttt{x:tm}$ to range $\gamma,\texttt{b:}\mathbf{block}\texttt{(x:term,ae\_v:aeq x x)}$ we simply construct a weakening substitution .. `b.1` that essentially renames `x`

to b.1 in M. The recursive call returns [γ,b:**block**(y:term,ae_v:aeq y y)⊢ D.. b.1 b.2]. Using it together with rule ae_l we build the final derivation.

3. Application case. If M is an application, we appeal twice to the induction hypothesis and build [γ ⊢ aeq (app (M1..)(M2..)) (app (M1..)(M2..))].

Beluga's type checker verifies that the manipulated derivations are well-formed and meaningful. For the recursive function reflG to constitute a proof, we also need to know that it is total, i.e., all cases are covered and the function is terminating. At the time of writing, Beluga provides a coverage checker that will automatically verify whether all cases are covered [12]; the fact that a given function is terminating can be verified automatically. Such arguments are usually direct and straightforward (see also [38] for a foundation restricting Beluga's language to primitive structural recursive functions).

4.2 Context Extensions with Alternative Declarations (G Version)

Extending the algorithmic equality case study to the polymorphic lambda-calculus to handle alternating assumptions is straightforward. For the specification of the algorithmic equality rules for polymorphic terms (written as aeq M N) and types (written as atp T S), we refer the kind reader to Appendix A.2. We can now establish reflexivity, transitivity, and symmetry of type equality in a straightforward way. These proofs are established in the context of schema atpG with

```
schema atpG = block (a:tp,v:atp a a);
```

Then, we add the additional case for type abstraction and type application to reflexivity, transitivity and symmetry of term equality. Similar issues as in the Twelf development arise; however the solution taken differs slightly.

1. Since the case for lambda-abstraction deals with term assumptions while the type abstraction introduces type assumptions, we need to specify *alternating* assumptions. This alternation of blocks is described by using + in Beluga's concrete syntax.

   ```
   schema aeqG = block (x:tm,u:aeq x x) + block(a:tp,v:atp a a);
   ```

2. The reflexivity, transitivity and symmetry properties for terms rely on the corresponding properties for types. As in Twelf, Beluga's type inference engine relies on subordination to justify instances of safe context weakening and strengthening.[2] When we call a function that requires a context γ of schema atpG but we have a context of schema aeqG, we check that every block b in schema atpG is a prefix of a block b′ in the schema aeqG and moreover additional declarations present in b′ are irrelevant to the declarations in b. The type checker therefore allows us to pass a context of schema aeqG whenever a context atpG is required.

The main difference between Twelf and Beluga with respect to the handling of contexts is which assumptions we track. Since Beluga is a functional language and allows users to implement a general variable case, the context characterizes only assumptions arising from the derivations we manipulate and matches our informal description given earlier. As a

[2]However, the operational semantics is not adapted and these programs are not executable; this would require us to add appropriate casting operations, which correspond to strengthening and weakening functions.

consequence, context subsumption checks can be often simpler, i.e., checking for a prefix usually suffices.

4.3 Non-Linear Context Extensions (G Version)

The completeness proof (Theorem 19) is implemented as a recursive function `ceqG` in Beluga. We first define the schema of the generalized context, following our informal development as follows:

```
schema daG = block (x:tm,u:aeq x x,v:deq x x);
```

as Beluga verifies that every block in `schema xaG =` **block** `(x:term,u:aeq x x)` is a prefix of a block in the context of schema `daG`. The observant reader will notice that our definition of `daG` does not match precisely the definition given in the ORBI file for Theorem 19, but we reordered the assumptions `aeq x x` and `deq x x` to enable context subsumption. As in Twelf, Beluga will verify whether the context required by the lemma is a proper sub-context of the context provided by the main theorem. In the type and implementation of the function `ceqG`, we indicate that the context γ is implicit in the actual implementation of the proof and will be reconstructed by using round brackets and writing the context declaration as `(`γ`:daG)` (see Fig. 6).

We explain here the three cases shown in the informal proof in the companion paper [17].

First, we consider the case where we have used an assumption from the context. Since the context γ consists of blocks such as **block** `(x:tm,ae_v:aeq x x,de_v:deq x x)`, we in fact want to match on the third element of such a block. This is written as `#p.3`... The type of `#p.3` is `deq (#p.1..) (#p.1..)`. Since our context always contains a block and the parameter variable `#p`.. describes such a block, we know that there exists a proof for `aeq (#p.1..) (#p.1..)`, which can be described by `#p.2`...

Second, we consider the case where we applied the reflexivity rule `de_r` as a last step. In this case, we need to refer to the reflexivity lemma we proved about algorithmic equality. To use the function `reflG`, which implements the reflexivity lemma for algorithmic equality, we need a context of schema `xaG`; however, the context used in the proof for `ceqG` is of

```
rec ceqG: (γ:daG)[γ ⊢ deq (T ..) (S ..)] → [γ ⊢ aeq (T ..) (S ..)] =
 fn e ⇒ case e of
| [γ ⊢ #p.3 ..] ⇒ [γ ⊢ #p.2 ..]                  % Assumption from context

| [γ ⊢ de_r ] ⇒ reflG [γ] [γ ⊢ _ ]               % Reflexivity

| [γ ⊢ de_t (D1 ..) (D2 ..)] ⇒                   % Transitivity
  let [γ ⊢ F2 ..] = ceqG [γ ⊢ D2 ..] in
  let [γ ⊢ F1 ..] = ceqG [γ ⊢ D1 ..] in
    trans [γ ⊢ F1 ..] [γ ⊢ F2 ..]

| [γ ⊢ de_s (D ..)] ⇒                            % Symmetry
  let [γ ⊢ F ..] = ceqG [γ ⊢ D ..] in
  sym [γ ⊢ F ..]

| [γ ⊢ de_l (λx.λu. D .. x u)] ⇒                 % Abstraction
  let [γ, b:block (x:tm,ae_v:aeq x x,de_v:deq x x) ⊢ F .. b.1 b.2] =
   ceqG [γ, b:block (x:tm,ae_v:aeq x x,de_v:deq x x) ⊢ D .. b.1 b.3]
  in
     [γ ⊢ ae_lam (λx.λv. F .. x v)]

| [γ ⊢ de_a (D2 ..) (D1 ..)] ⇒                   % Application
    let [γ ⊢ F1 ..] = ceqG [γ ⊢ D1 ..] in
    let [γ ⊢ F2 ..] = ceqG [γ ⊢ D2 ..] in
      [γ ⊢ ae_a (F1 ..) (F2 ..)] ;
```

Fig. 6 Completeness of algorithmic equality in Beluga

schema daG and we rely on context subsumption to justify passing a context daG in place of a context xaG. The cases for transitivity and symmetry are similar.

Third, we consider the case for de_l, i.e., the case for lambda-abstractions. We extend the context with the new variable declarations and make a recursive call on the sub-derivation [γ, b:**block**(x:tm,ae_v:aeq x x,de_v:deq x x)⊢ D.. b.1 b.3]. Declaration weakening (*d-wk*) used in the informal proof [17] is built-in. In the pattern, the derivation D has type [γ,x:tm,ae_v:aeq x x⊢ deq (M.. x)(N.. x)]. We hence construct a weakening substitution .. b.1 b.3, which allows us to move D to the context γ, b:**block**(x:tm,ae_v:aeq x x,de_v:deq x x). The result of the recursive call is a derivation F, where F only depends on x:term and u:aeq x x. In the on-paper proof we refer to declaration strengthening (*d-str*) to justify that F cannot depend on de_v assumptions. In Beluga, the programmer uses strengthening by stating which assumptions F can depend on. The coverage checker will then subsequently rely on subordination to verify that the restricted case is sufficient and no other cases have been forgotten. Subordination allows us to verify that indeed assumptions of type de_v:deq x x cannot be used in establishing a proof for aeq (M.. b.1) (N.. b.1). Finally, we use F to assemble the final result ae_l (λx.λv. F.. x v).

We conclude this example with a few observations: The statement of the theorem is directly and succinctly represented in Beluga using contextual types and contextual objects. Every case in the on-paper proof corresponds directly to a case in the implementation of the recursive function. Type reconstruction is used to reconstruct implicit type arguments and infer the type of free contextual variables that occur in patterns. This is crucial to achieve a palatable source language. Weakening and strengthening are supported in Beluga through the typing rules and on the level of context variables and context schemas using context subsumption. If schema W is a prefix of a schema W', then we can always use a context of schema W' in place of a context of schema W provided that the additional assumptions present in W are irrelevant.

4.4 Order (G Version)

The order of assumptions in a context is important in Beluga. However, sometimes the need to reorder assumptions arises, as is illustrated in the proof of the substitution lemma for algorithmic equality. As in Twelf this kind of proof (Theorem 23) does not come for free in Beluga (see Fig. 7). In Beluga, exchanging the order of assumptions in the context is managed via substitutions. In the case ae_l, we consider D.. b y ae_y where D is a subderivation in the context γ,b:**block**(x:tm,ae_x:aeq x x),y:tm,ae_y:aeq y y). To appeal to the IH, we recurse on the sub-derivation D.. d d'.1 d'.2 in the extended context γ,d':**block**(y:tm,ae_y:aeq y y),d:**block**(x:tm,ae_x:aeq x x). The permutation substitution .. d d'.1 d'.2 provides a mapping from the variables in the context γ,b:**block**(x:tm,ae_x:aeq x x),y:tm,ae_y:aeq y y to variables in the context γ,d':**block**(y:tm,ae_y:aeq y y),d:**block**(x:tm,ae_x:aeq x x).

This proof also highlights another interesting aspect of Beluga: it allows recursion over the context. There are two base cases in this proof. First, we consider the derivation aeq x x where we want to replace the variable x by the term N1 in M1 and by N2 in M2. Since we have a derivation d' describing [γ ⊢ aeq (N1..)(N2..)] by assumption, we simply return d'.

Second, we want to consider the case where we have a variable from the context, but this variable is not x as in [γ,b:**block** (x:tm,ae_v:aeq x x)⊢ #p.2..]. We know that the parameter variable #p.. describes some assumption from γ and in particular,

```
schema xaG = block (x:tm,ae_v:aeq x x);

rec subst: (γ:xaG)
        [γ, b:block (x:tm,ae_v:aeq x x) ⊢ aeq (M1 .. b.1) (M2 .. b.1)]
      → [γ ⊢ aeq (N1 ..) (N2 ..)]
      → [γ ⊢ aeq (M1 .. (N1 ..)) (M2 .. (N2 ..)) ] =
fn d ⇒ fn d' ⇒ case d of
| [γ, b:block (x:tm,ae_v:aeq x x) ⊢ b.2] ⇒ d'

| [γ, b:block (x:tm,ae_v:aeq x x) ⊢ #p.2 .. ] ⇒ [γ ⊢ #p.2 ..]

| [γ, b:block (x:tm,ae_x:aeq x x) ⊢ ae_l (λy.λae_y. D .. b y ae_y)] ⇒
  let [γ ⊢ D' .. ] = d' in
  let [γ, d':block (y:tm,ae_y:aeq y y) ⊢ F .. d'.1 d'.2] =
    subst [γ, d':block (y:tm,ae_y:aeq y y),
               d :block (x:tm,ae_x:aeq x x) ⊢ D .. d d'.1 d'.2 ]
          [γ, d' ⊢ D' .. ] in
  [γ ⊢ ae_l (λy.λae_y. F .. y ae_y)]

| [γ, b:block (x:tm,ae_x:aeq x x) ⊢ ae_a (D1 .. b) (D2 .. b)] ⇒
  let [γ ⊢ F1 ..] = subst [γ, b:block (x:tm,ae_x:aeq x x) ⊢ D1 .. b] d' in
  let [γ ⊢ F2 ..] = subst [γ, b:block (x:tm,ae_x:aeq x x) ⊢ D2 .. b] d' in
      [γ ⊢ ae_a (F1 ..) (F2 ..)];
```

Fig. 7 Pairwise substitution Lemma in Beluga

x cannot occur in it, since otherwise the context would be ill-formed. The substitution .. associated with #p.2 is a (weakening) substitution mapping variables from context γ to variables in context γ,b:**block** (x:tm,ae_v:aeq x x). We therefore simply return [$\gamma \vdash$ #p.2..].

4.5 Uniqueness (G Version)

We discuss next the implementation of the proof of type uniqueness for simply typed lambda-terms. The specification of terms together with their typing rules can be found in Appendix A.3. To express the relation between a term M and its corresponding type T, we write oft M T. Since Beluga does not support equality types, we implement equality using a LF type family eq as we did in the Twelf section. The uniqueness proof (see Fig. 8) itself is mostly straightforward (see also [37] for a longer discussion). The only interesting case arises when we encounter a variable of type tm from the context. Since our context consists of blocks containing variables of type term and assumptions oftx T1, we pattern match on [$\gamma \vdash$ #p.2..], i.e., we project out the second argument of the block. We hence know that [$\gamma \vdash$ #p.2..] has type [$\gamma \vdash$ oft(#p.1..) T], because the parameter variable #p has type [$\gamma \vdash$ **block**(x:tm,u:oftx T)]. We also know that the second input, called f, to the function unique has type [$\gamma \vdash$ oft(#p.1..) S]. By inversion on f, we know that the only possible object that can inhabit this type is [$\gamma \vdash$ #p.2..] and therefore S must be identical to T. Moreover, #q denotes the same block as #p.

We note that some of the issues concerning the definition of equality via reflexivity in Twelf do not arise in Beluga; Beluga's proof is implemented as a function using pattern matching instead of relations; as we pattern match we learn more about the given derivations and information flows as expected.

4.6 Substitution (G Version)

Beluga enjoys the usual substitution property for parametric and hypothetical derivations for free. Consider the proof of type preservation for the simply-typed lambda-calculus

```
schema xtG = some [t:tp] block (x:tm,u:oft x t);

rec unique: (γ:xtG)[γ ⊢ oft(M..) T] → [γ ⊢ oft(M..) S] → [ ⊢ eq T S] =
fn d ⇒ fn f ⇒ case d of
| [γ ⊢ of_app (D1..) (D2..)] ⇒
  let [γ ⊢ of_app (F1..) (F2..)] = f in
  let [ ⊢ e_ref] = unique [γ ⊢ D1..] [γ ⊢ F1..] in
    [ ⊢ e_ref]

| [γ ⊢ of_lam (λx.λu. D.. x u)] ⇒
  let [γ ⊢ of_lam (λx.λu. F.. x u)] = f in
  let [ ⊢ e_ref] = unique [γ, b:block (x:tm,t:oft x _) ⊢D.. b.1 b.2]
                          [γ, b ⊢ F.. b.1 b.2] in
   [ ⊢ e_ref]

| [γ ⊢ #p.2..] ⇒ % d :oft #p.1 T
  let [γ ⊢ #q.2..] = f in % f :oft #p.1 S
    [ ⊢ e_ref];
```

Fig. 8 Type uniqueness in Beluga

with parallel reductions (Theorem 27): when M steps to N and M has type A then N has the same type A. Since we allow reductions underneath an abstraction, we may step through terms containing variables. We therefore allow expressions to depend on the context γ.

We only show the case for beta-reduction in Fig. 9. Consider we have used the clause pr_b:red (app (lam λx.M.. x)(N..))(M..(N..)) to step. Then we have as assumption d:[γ ⊢ oft(app (lam A (λx. M.. x))(N..)) (arr A B). By inversion, we know that d must stand for a derivation [γ ⊢ of_a (of_l λx.λu. D1.. x u)(D2..)] where D1 stands for oft(M.. x) B in the extended context γ,x:tm, u:oftx A. We also know that D2 describes a derivation oft(N..) A. By the IH (recursive call) on D2, we obtain a derivation F2:oft(N..) A. By the IH (recursive call) on D1, we get a derivation F1:oft(M.. x) B in the context γ,b:**block** (x:tm,of_x:oftx A). We now want to substitute for x the term N, and for the derivation oftx A the derivation F2. This is achieved by applying to F1 the substitution .. _(F2..). Since in the program above we do not have the name N available, we write an underscore and let Beluga's type reconstruction algorithm infer the appropriate name.

4.7 Mechanizing Properties in Beluga$^{\mu}$ Using Recursive Types and Substitutions

So far, we have concentrated on mechanizing proofs in Beluga only relying on a fragment which contains (dependent) functions and applications, and pattern matching on contextual objects and contexts. However, Beluga$^{\mu}$ [8] also supports indexed recursive datatypes and

```
schema xrtG = some [t:tp] block (x:tm,of_x:oft x t,pr_x:pr x x);

rec tps:(γ:xrtG)[γ ⊢ pr(M..) (N..)] → [γ ⊢ oft(M..) A] → [γ ⊢ oft(N..) A] =
fn f ⇒ fn d ⇒ case f of
| [γ ⊢ #p.3. ] ⇒ d

| [γ ⊢ pr_b (λx.λpr_x. R1.. x pr_x) (R2.. ) ] ⇒
  let [γ ⊢ of_a (of_l (λx.λof_x. D1.. x of_x)) (D2..) ] = d in
  let [γ, b:block (x:tm,of_x:oft x T,pr_v:pr x x) ⊢F1.. b.1 b.2] =
    tps [γ, b:block (x:tm,of_x:oft x _,pr_v:pr x x) ⊢R1.. b.1 b.3]
        [γ, b ⊢ D1.. b.1 b.2] in
  let [γ ⊢ F2.. ] = tps [γ ⊢ R2..] [γ ⊢ D2..] in
    [γ ⊢ F1.. _ (F2..)]                % use substitution lemma directly
```

Fig. 9 Type preservation for parallel reductions in Beluga

first-class substitutions [9]. This allows us to define predicates about contexts and contextual objects, relate contexts via substitutions, and more importantly prove properties about them inductively. The additional expressiveness gained allows us to explicitly express promotion lemmas and proofs using context relations (R version).

4.7.1 Promotion

In the previous sections, we have taken advantage of built-in context subsumption in checking Beluga proofs. If schema W is a prefix of a schema W', then we can always use a context of schema W' in place of a context of schema W. This enables safe instances of context weakening and context strengthening and allows us to use the reflexivity lemma (`reflG`) in the completeness proof for algorithmic equality (`ceqG`), although the function `reflG` requires a context `xaG`, while the function `ceqG` provides a context `daG`. While such uses of context subsumption make proofs short and compact, and are convenient for users, one might ask whether one can avoid referring to context subsumption and make the required reasoning explicit, namely context weakening and strengthening. In other words, we eliminate context subsumption by referring to promotion lemmas such as Lemma 18.

The key to expressing context weakening and strengthening is the ability to relate two contexts via a substitution. In Beluga$^{\mu}$, we can describe context relations using *inductive datatypes* as a relation between context ϕ, context γ and a substitution σ that maps variables from ϕ to the context γ, formally $\gamma \vdash \sigma : \phi$, as follows:

```
datatype   Ctx_R : {φ:xaG}{γ:daG}{σ:[γ ⊢ φ]} ctype =
| Nil : Ctx_R [] [] [ ⊢  ^ ]
| Cons: Ctx_R [φ] [γ] [γ ⊢ σ]
     → Ctx_R [φ, b:block(x:tm,u:aeq x x)]
             [γ, b:block(x:tm,u:aeq x x,v:deq x x)] [γ,b ⊢ σ <b.1,b.2>]
;
```

We use first-class substitution variables σ of type $[\gamma \vdash \phi]$ [9] to provide a *witness* on how to move from the context ϕ to the context γ. We emphasize that the datatype `Ctx_R` is *not* an LF type, but instead an indexed recursive type that lives on the reasoning level. It corresponds to inductive definitions in other systems. This is indicated by declaring `Ctx_R` to be a `ctype`, which is indexed by the context ϕ`:xaG`, the context γ`:daG` and the substitution $\sigma : [\gamma \vdash \phi]$. In our example, the substitution σ is simply a variable substitution that weakens terms declared in the context ϕ to be meaningful in γ.

The relation `Ctx_R` is defined inductively based on the structure of contexts ϕ and γ. If ϕ and γ denote the empty context, the empty substitution, written as `^`, acts as a witness. If σ relates the context ϕ and γ, then the context ϕ`,b:`**block**`(x:tm,u:aeq x x)` is related to the context γ`,b:`**block**`(x:tm,u:aeq x x,v:deq x x)` by the substitution[3] σ`<b.1,b.2>` where we supply a tuple `<b.1,b.2>` for the declaration **block**`(x:tm,u:aeq x x)`.

[3] We take here some liberty and write simply σ; in the theoretical foundation, substitution variables do not occur by themselves; similar to meta-variables and parameter variables they are associated with a postponed substitution.

Given a term M in the term context ϕ and Ctx_R [ϕ] [γ] [$\gamma \vdash \sigma$], where σ is a weakening substitution from ϕ to γ, we can always weaken M by σ to obtain [$\gamma \vdash$ M σ]. This is encoded in the datatype definition below:

```
datatype  StrTm : (γ:daG)[γ ⊢ tm] →  ctype =
| StrTm : {σ:[γ ⊢ φ]} Ctx_R [φ] [γ] [γ ⊢ σ]
          → {M:[φ ⊢ tm]} StrTm [γ ⊢ M σ];
```

We call the datatype StrTm, because we want to show that given a term [$\gamma \vdash$ M..] there always exists a StrTm [$\gamma \vdash$ M..], i.e., there exists a term context γ, a weakening substitution σ and a term [$\phi \vdash$ N..] such that N σ is equal to M. Here N is the strengthened term. This strengthening for terms is implemented by a recursive function of type:

```
str_tm : {γ:daG}{M:[γ ⊢ tm]} StrTm [γ ⊢ M .. ]
```

The proof is straightforward, but because we have defined a context relation using a recursive data-type instead of a function, which when given a context ϕ of schema xaG returns a term context γ together with a weakening substitution σ, we need two additional lemmas:[4] we need to show that the context relation is deterministic and whenever we have a context ϕ of schema daG, there exists a term context γ of schema xaG. We can then implement promotion of reflexivity to the context daG as follows:

```
rec  promote_refl: {γ:daG}{M:[γ ⊢ tm]} [γ ⊢ aeq (M ..) (M ..)] =
λ□γ ⇒ λ□M ⇒
let  StrTm  [γ ⊢ σ] cr [φ ⊢ N ..] = str_tm [γ] [γ ⊢ M ..] in
let  [φ ⊢ D ..] = reflG [φ] [φ ⊢ N ..] in
  [γ ⊢ D σ];
```

Given a term M in the context γ, we first strengthen it, obtaining term N in the strengthened context ϕ together with the substitution σ that maps variables in ϕ to variables in γ. We now call reflG with the term context ϕ it requires together with the term N and obtain a proof [$\phi \vdash$ D..] standing for [$\phi \vdash$ aeq (N..) (N..)]. Context weakening, i.e., weakening D such that it makes sense in the original context γ, is accomplished by simply applying the substitution σ.

The full development of the completeness of algorithmic equality with explicit context strengthening and promotion is available online (see eq-proof-promotion.bel). This development also highlights how convenient built-in support for context subsumption is in Beluga; our proof development using context subsumption (115 lines of code) is approximately a third of the proof development using explicit strengthening and promotion lemmas (318 lines of code).

4.7.2 Proofs Using Context Relations

We have seen that Beluga elegantly supports proofs using generalized contexts. Here we exploit the ability to define context relations in Beluga$^{\mu}$ to implement proofs using con-

[4] Beluga$^{\mu}$ does not support functions in types at this point.

text relations (R version). To illustrate the idea, we go back to the reflexivity proof for algorithmic equality.

```
datatype  Ctx_xaR : {φ:xG}{ψ:xaG}{σ:[ψ ⊢ φ]} ctype =
| Nil_xa : Ctx_xaR [] [] [ ⊢  ^ ]
| Cons_xa: Ctx_xaR [φ] [ψ] [ψ ⊢ σ ]
     → Ctx_xaR [φ,x:tm] [ψ, b:block(x:tm,u:aeq x x)] [ψ,b ⊢ σ b.1 ];
```

We again use first-class substitution variables σ of type `[ψ ⊢ φ]` to move from the context ϕ to the context ψ. If σ relates ϕ and ψ, then the substitution `σb.1` relates context ϕ`,x:tm` to ψ`,b:`**block**`(x:tm,u:aeq x x)` via constructor `Cons_xa`.

We show the proof `reflR` using context relations in Fig. 10. It is a straightforward recursive program of type

```
{φ:xG}{M:[φ ⊢ tm]} Ctx_xaR [φ] [ψ] [ψ ⊢σ] → [ψ ⊢ aeq (M σ) (M σ)]
```

which can be read as: For all contexts ϕ and ψ that have schema `xG` and `xaG` respectively, if we have a substitution σ such that $\psi \vdash \sigma : \phi$, then for all terms `M` depending on ϕ, we have a proof that `[ψ ⊢ aeq (M σ) (M σ)]`. We note that because the term `M` depends only on the context ϕ, we explicitly weaken it by applying σ to move it to the context ψ.

The proof is implemented as a recursive function over the term `M`. We again consider three possible cases. If we have an application `[φ ⊢ app (M1..)(M2..)]`, we appeal to the induction hypothesis on `[φ ⊢ M1..]` and `[φ ⊢ M2..]` by making a recursive call. Since the context ϕ and the context ψ do not change, we can simply make the recursive call on `[φ ⊢ M1..]` and `[φ ⊢ M2.. ]` respectively using the relation `cr`.

```
rec  ctx_membership: {#p:[φ ⊢ tm] } Ctx_xaR [φ] [ψ] [ψ ⊢ σ] →
                      [ψ ⊢ aeq (#p σ) (#p σ)] =
λ□#p ⇒ fn  cr ⇒  let  (cr : Ctx_xaR [φ] [ψ] [ψ ⊢ σ]) = cr in
case  [φ ⊢ #p..] of
 | [φ, x:tm ⊢  x] ⇒
   let  Cons_xa cr' = cr in
   let  (cr' : Ctx_xaR [φ] [ψ] [ψ ⊢ σ]) = cr' in
     [ψ,b:block (x:tm,u:aeq x x) ⊢  b.2]

| [φ, x:tm ⊢ #p..] ⇒
  let  Cons_xa cr' = cr in
  let  [ψ ⊢  E..] =  ctx_membership [φ ⊢ #p..] cr'  in
       [ψ,b:block (x:tm,u:aeq x x) ⊢  E.. ];

rec  reflR: {φ:xG}{M:[φ ⊢ tm]} Ctx_xaR [φ] [ψ] [ψ ⊢ σ] →
             [ψ ⊢ aeq (M σ) (M σ)] =
λ□φ ⇒ λ□M ⇒ fn cr ⇒ case  [φ ⊢ M..] of
| [φ ⊢  #p..] ⇒ ctx_membership [φ ⊢ #p..] cr
| [φ ⊢ app (M.. ) (N..)] ⇒
  let  [ψ ⊢ D1.. ] = reflR [φ] [φ ⊢ M..] cr in
  let  [ψ ⊢ D2.. ] = reflR [φ] [φ ⊢ N..] cr in
    [ψ ⊢ ae_a (D1..) (D2..) ]
| [φ ⊢ lam λx.M.. x] ⇒
  let  [ψ, b:block(x:tm,u:aeq x x) ⊢  D.. b.1 b.2] =
       reflR [φ,x:tm] [φ,x:tm ⊢ M.. x] (Cons_xa cr) in
  [ψ ⊢ ae_l (λx.λu. D.. x u)];
```

Fig. 10 Reflexivity of algorithmic equality in Beluga$^{\mu}$ (R version)

When we have [$\phi \vdash$ lam λx.M.. x], we want to appeal to the induction hypothesis on [ϕ,x:tm $\vdash$ M.. x]. To make the recursive call, we also need a witness relating the context [ϕ,x:tm $\vdash$ M.. x] to the context [ψ,b:**block**(x:tm,u:aeq x x)]. Recall that cr stands for Ctx_xaR [ϕ] [ψ] [$\psi \vdash \sigma$]. Therefore, by Cons_xa, we know there exists a witness for Ctx_xaR [ϕ,x:tm] [ψ,b:**block**(x:tm,u:aeq x x)] [ψ,b $\vdash$ σ b.1] and we appeal to the IH by reflR [ϕ,x:tm] [ϕ,x:tm $\vdash$ M.. x] (Cons_xa cr).

Finally, we take a close look at the variable case. For clarity, we follow the outline of the proof from the companion paper [17] and factor out the context membership lemma (Lemma 6, see Section 2.2). We distinguish two different cases depending on the position of the variable in the context by pattern matching on the shape of ϕ. If [ϕ,x:tm.x], then we inspect the context relation cr. We note that pattern matching forces the original context ϕ to be ϕ,x:tm. By pattern matching on cr', we observe that there exists a relation cr', such that Ctx_xaR [ϕ] [ψ] [$\psi \vdash \sigma$]. Moreover, $\psi = \psi$, b:**block**(x:tm,u:aeq x x) and $\sigma = \sigma$ b.1 where the left hand side denotes the original context and substitution, while the right hand side shows the context and substitution refinement after pattern matching. We must show that there exists a proof of aeq x x in the context ψ,b:**block**(x:tm,u:aeq x x). This is simply b.2.

If instead [ϕ,x:tm$\vdash$#p..], we have a variable other than x. In this case, we also observe that the contexts ϕ,x:tm and ψ,b:**block**(x:tm,u:aeq x x) are related by the substitution σ b.1 since we have the assumption cr. By inversion on Cons_xa we know that cr' relates ϕ and ψ by the substitution σ. We now appeal to the induction hypothesis making the recursive call ctx_membership [$\phi \vdash$#p..] cr'. This is a valid recursive call, since the context ϕ is smaller than the original context. We then rely on declaration weakening to obtain the desired result.

Substitution variables account, in principle, for a concise representation of proofs using context relations; they are particularly useful to elegantly express proofs by logical relations [10] where substitutions play an essential role. In the completeness proof of algorithmic equality, they make possible the implementation of the proof using context relations. However, the overhead of defining context and equivalence relations and proving that the defined relations are total functions is substantial in the R version of the completeness proof of algorithmic equality.[5]

5 The Two-Level Approach

In this section we briefly review the two-level architecture that Hybrid and Abella share, referring to [16] and [22] for more extensive explanations.

In this approach we distinguish between a simple *specification logic* (SL), where we specify (and execute) our judgments, and a *reasoning* logic (RL), endowed with some form of induction/recursion, where we conduct our proofs. Abella and Hybrid use different RLs, the former being the $\mathcal{G}$ logic [21], an intuitionistic first-order sequent calculus with a fixed-point approach to (co)induction and the nabla quantifier ∇, and the latter being the Calculus of Inductive Constructions (CIC) as implemented by Coq. On the other hand, they basically share the same SL, a (minimal) calculus for hereditary Harrop formulas, as we explain

[5]The completeness proof of algorithmic equality (G version) in Beluga is 115 lines of code compared to its R version that takes 427 lines of code.

$$\frac{}{\Sigma;\Gamma \longrightarrow_{\mathscr{C}} \top}\top_R \qquad \frac{\Sigma;\Gamma \longrightarrow_{\mathscr{C}} G_1 \qquad \Sigma;\Gamma \longrightarrow_{\mathscr{C}} G_2}{\Sigma;\Gamma \longrightarrow_{\mathscr{C}} G_1 \wedge G_2}\wedge_R$$

$$\frac{(\Sigma,a);\Gamma \longrightarrow_{\mathscr{C}} G[a/x]}{\Sigma;\Gamma \longrightarrow_{\mathscr{C}} \forall x.\, G}\forall_R \qquad \frac{\Sigma;\Gamma \longrightarrow_{\mathscr{C}} G[t/x]}{\Sigma;\Gamma \longrightarrow_{\mathscr{C}} \exists x.\, G}\exists_R \qquad \frac{\Sigma;(\Gamma,A) \longrightarrow_{\mathscr{C}} G}{\Sigma;\Gamma \longrightarrow_{\mathscr{C}} A \supset G}\supset_R$$

$$\frac{}{\Sigma;(\Gamma,A) \longrightarrow_{\mathscr{C}} A}\text{init} \qquad \frac{\Sigma;\Gamma \longrightarrow_{\mathscr{C}} [\mathbf{t}/\mathbf{x}]G}{\Sigma;\Gamma \longrightarrow_{\mathscr{C}} A}\text{bc}$$

The rule bc has the proviso that $\forall \mathbf{x}\ (G \supset A') \in \mathscr{C}, [\mathbf{t}/\mathbf{x}]A' = A$.

Fig. 11 A minimal sequent calculus with backchaining

below. We use $\mathbf{x}$ to mean a vector of variables $x_1, \ldots, x_n$ in $\forall \mathbf{x}\ (G \supset A)$, where A is an atomic formula. We extend our vector notation to substitutions writing $[\mathbf{t}/\mathbf{x}]G$ as a shorthand for $[t_1/x_1,\ \ldots,\ t_n/x_n]G$.

$$\begin{array}{rrcl} \textit{Clauses} & C & ::= & \forall \mathbf{x}\ (G \supset A) \\ \textit{Goals} & G & ::= & \top \mid A \mid G_1 \wedge G_2 \mid A \supset G \mid \forall x.\, G \mid \exists x.\, G \\ \textit{Context} & \Gamma & ::= & \cdot \mid \Gamma, A \\ \textit{Signature} & \Sigma & ::= & \cdot \mid \Sigma, a \end{array}$$

Note that this SL is less powerful than the logical framework LF, the specification language of Twelf and Beluga, as it restricts implications to $A \supset G$, i.e., we only make atomic assumptions. In this sense we talk about *second-order* hereditary Harrop formulas, where by *order* we mean the implicational complexity of a clause and not the domain of quantification.[6]

This language is simple enough that its sequent calculus is analogous to a logic programming interpreter. In fact, we can write inference rules so that the only left rule is similar to Prolog's backchaining. Sequents have the form $\Sigma;\ \Gamma \longrightarrow_{\mathscr{C}} G$, where Σ is the current signature of eigenvariables, and we distinguish clauses belonging to a static database $\mathcal{C}$ from formulas in Γ. Intuitively, the former represent the specification, and the latter are the atoms introduced by the right implication rule. The rules for this logic are given in Fig. 11.

Note that in the proof-theoretic setting, as opposed to the type-theoretic one, a declaration block in a context such as `block (x:tm;u:aeq x x)` is split into two parts: a *generic* part, which essentially only records that `x` is indeed a (fresh) eigenvariable of sort `tm` and a *hypothetical* one, which states the relevant assumption for the given eigenvariable. Hence the separation between Σ and Γ, although Hybrid and Abella will differ on how genericity is realized. Genericity, in fact, is a property of the quantification structure of the RL, while hypothetical judgments are made possible by encapsulation inside the SL.

[6] The two-level architecture makes possible, indeed it encourages, to plug-in a different SL; in fact, Abella 2.0 [47] features an n-ary version of Harrop formulas, where assumptions can have arbitrary complexity. However, because all the benchmarks that we study here are second-order only, we take the liberty of describing here the second-order version of the SL. Note that the scripts described in the Abella section run under Abella 2.0 and that some of the differences are explained thereby.

$$
\begin{array}{ll}
\texttt{s\_t}: & \{\Gamma \vdash_{n+1} \top\} \\
\texttt{s\_and}: & \{\Gamma \vdash_n G_1\} \to \{\Gamma \vdash_n G_2\} \to \{\Gamma \vdash_{n+1} G_1 \wedge G_2\} \\
\texttt{s\_all}: & \mathbf{Q}x.\, \{\Gamma \vdash_n G\, x\} \to \{\Gamma \vdash_{n+1} \forall_\tau (\lambda x.\, G\, x)\} \\
\texttt{s\_ex}: & \{\Gamma \vdash_n G\, M\} \to \{\Gamma \vdash_{n+1} \exists_\tau (\lambda x.\, G\, x)\} \\
\texttt{s\_imp}: & \{A :: \Gamma \vdash_n G\} \to \{\Gamma \vdash_{n+1} A \supset G\} \\
\texttt{s\_init}: & A \in \Gamma \to \{\Gamma \vdash_{n+1} \langle A \rangle\} \\
\texttt{s\_bc}: & \texttt{prog}\, A\, G \to \{\Gamma \vdash_n G\} \to \{\Gamma \vdash_{n+1} \langle A \rangle\}
\end{array}
$$

Fig. 12 Encoding of specification logic

The SL and object level judgments are themselves encoded in the RL as *inductive* definitions with the following types:

```
seq  : atm list → nat → o → prop
prog : atm → o → prop
```

where **prop** is the kind of propositions in the RL, `atm` represents atomic predicates of the object-language, and `o` is the type of SL propositions. Natural numbers (`nat`) decorate sequents, in order to reason by (complete) induction on the *height* of a proof. The logic connectives have sorts:

$$
\top : \texttt{o} \mid \langle \_ \rangle : \texttt{atm} \to \texttt{o} \mid \wedge : \texttt{o} \to \texttt{o} \to \texttt{o} \mid \supset : \texttt{atm} \to \texttt{o} \to \texttt{o} \mid
$$
$$
\forall_\tau : (\tau \to \texttt{o}) \to \texttt{o} \mid \exists_\tau : (\tau \to \texttt{o}) \to \texttt{o}
$$

where $\langle \_ \rangle$ coerces atoms into SL propositions.[7] In the definition of the SL (see Fig. 12), we write $\{\Gamma \vdash_n G\}$ to pretty-print the `seq` definition, to follow Abella's concrete syntax and increase readability in the next sections, and we overload the usual logical symbols, e.g., the clause `s_imp` in Fig. 12 is a pretty-printed version of the following:

```
seq (A::Γ) N G → seq Γ (N+1) (A ⊃ G)
```

Free variables in inductive definitions and also in statements of theorems are implicitly universally quantified at the top-level of each clause or statement. Contexts are represented as lists,[8] the :: operator represents cons, and the $\in$ operator list membership. The first five clauses of the definition directly encode the introduction rules of a sequent calculus for this logic. The rule `s_all` maps the SL universal quantifier to the related universal in the RL (here ambiguously denoted by **Q**), which in the case of Hybrid will be Coq's dependent product and in the case of Abella the ∇ quantifier. In `s_ex` we rely on meta-logic unification to instantiate the logic variable M.[9] In the last two rules, atoms are provable either by assumption or via *backchaining* over Prolog-like rules, the `prog` predicate, which encodes the aforementioned database $\mathcal{C}$ of the clauses constituting the "program" under consideration. If we look at the SL as a variant (with embedded implications) of the standard Prolog vanilla meta-interpreter, `prog` would correspond to Prolog's built-in predicate

[7] The τ argument to quantifiers ranges in Abella over any sort *not* containing `o`, while in Hybrid it is restricted to `expr`, the built-in sort of object-level expressions. We discuss this issue further in Section 8.

[8] Using lists as the data-structure underlying contexts is not written in stone. In fact, the Isabelle/HOL version of Hybrid uses sets [16] and Abella may also offer multi-sets in the near future.

[9] Abella does not support existentials goals, and, to overcome this, the user needs to resort to some indirect encoding, as exemplified in [29].

`clause/2`. The sequent calculus/meta-interpreter is parametric in those clauses and uses the RL underlying unification to do the matching required by the `s_bc` rule.

For example, clause `ae_l` in Appendix A is encoded as the following `prog` clause, where we do *not* explicitly encode the outermost SL universal quantifier, but we short-circuit it with the one of the meta-logic, hence omitted:

```
prog (aeq (lam (λx.M x)) (lam (λx.N x)))
     (∀tm (λx.aeq x x ⊃ <aeq (M x) (N x)>)).
```

In the rest of this section we will use a more readable "prog-less" syntax closer to that of ORBI, but we refer to Fig. 13 in the following Hybrid section for the `prog` clauses encoding benchmark A.1. As explained in the companion paper, ORBI specifications need to be massaged to fit the proof-theoretic view at the level of inference rules, schemas, and context relations. This is mostly due to the asymmetry between declarations that give typing (sorting) information, e.g., `tm:type`, and those expressing judgments, e.g., `aeq:tm -> tm -> type`. The logic typically does not know that `M:tm` entails is_tm *M* holding at the level of judgments, because the language of *sorts* tends not to be reflected in the proof-theory. Therefore, for the type `tm`, we introduce the predicate `is_tm` with the following clauses:

```
tm_a: is_tm M → is_tm N → is_tm (app M N).
tm_l: ({x} is_tm x → is_tm (M x)) → is_tm (lam (λx.M x)).
```

These well-formedness rules have two important applications:

1. to provide induction principles at the level of the syntax as for example required in the proof of admissibility of reflexivity (Theorem 7); see e.g., H-Theorem 2 and A-Theorem 2, where we prefix theorems with H (resp. A) writing H-Theorem (resp. A-Theorem) if we are referring to Hybrid's or Abella's encoding;
2. to prove preservation of well formedness results such as H-Theorem 30 and A-Theorem 12.[10]

In fact, with respect to the latter, we assumed in our presentation in the companion paper that whenever a judgment was provable, the terms in it were well-formed. In the two-level approach, such statements need to be proved explicitly, and to do so, annotations are added to inference rule encodings. In Hybrid and Abella, for instance, clauses `de_l` and `de_r` will have the form:

```
de_l: ({x} is_tm x → deq x x → deq (M x) (N x))
        → deq (lam (λx.M x)) (lam (λx.N x)).
de_r: is_tm M → deq M M.
```

In order to avoid clutter, annotations that are not needed for adequacy are left out; this is the case for the `ae_l` rule above. We will take the same approach when specifying context relations.

[10] We refer to these properties as "internal adequacy", as opposed to "external adequacy"; the latter refers to the bijection that should hold between the mathematical presentation of a OL and its encoding in a logical framework. See the discussion in [16, 22, 34] for Twelf, Abella and Hybrid respectively.

A small set of structural rules of the SL is proved once and for all, and then used to reason about object-languages:

SL-Theorem 1 (**Structural Properties**)

(*a*) Height weakening: $\{\Gamma \vdash_n G\} \to n < m \to \{\Gamma \vdash_m G\}$
(*b*) Antecedent weakening: $\{\Gamma \vdash_n G\} \to \Gamma \subseteq \Gamma' \to \{\Gamma' \vdash_n G\}$
(*c*) Atomic cut: $\{A :: \Gamma \vdash_n G\} \to \{\Gamma \vdash_m \langle A \rangle\} \to \{\Gamma \vdash_{n+m} G\}$
(*d*) Specialization: $\{\Gamma \vdash_n \forall_\tau (\lambda x.G\ x)\} \to \{\Gamma \vdash_n G\ t\}$

Antecedent weakening is defined "set-theoretically", i.e., $\Gamma \subseteq \Gamma'$ iff $\forall A.\ A \in \Gamma \to A \in \Gamma'$, and, as such, it subsumes contraction and in particular:

$$\textit{Exchange}:\ \Gamma \subseteq \Gamma' \wedge \Gamma' \subseteq \Gamma \to \{\Gamma \vdash_n G\} \text{ iff } \{\Gamma' \vdash_n G\}.$$

Declaration strengthening *d-str* is currently unsupported, and thus it is established on a case-by-case basis, typically with inductive proofs, although it is conceivable that it could be proven once and for all by enriching the `prog` clause with a dependency analysis, akin to Twelf and Beluga's subordination. Context strengthening *c-str* also requires separate proofs and its automation seems more complex.

As a consequence of the proof-theoretic separation between generic and hypothetical judgments, the *substitution* property is realized by a combination of cut and specialization. See for example the proof of A-Theorem 22, viz. the `pr_b` case. Specialization is just function application in Coq, while it is a rather deep property of the nabla quantifier and SL-derivability [22] in Abella.

Finally, context schemas/relations are realized by *inductive definitions* of certain predicates/relations in the RL and will follow the syntax of the ambient logic, as we will see for example at pages 30 and 43.

6 Mechanization in Hybrid

As mentioned, in the version of Hybrid used in this paper, the RL is Coq's Calculus of Inductive Constructions. Hybrid is implemented as a Coq library. We use a pretty-printed version of Coq concrete syntax, which includes some notation already presented. For example, `Prop` is the type of propositions of the RL, corresponding to `prop` in the previous section. The type `o` from the previous section is defined here as the following Coq inductive type:

$$\begin{aligned}\texttt{Inductive o} := \texttt{tt} : \texttt{o} \mid \langle \_ \rangle : \texttt{atm} \to \texttt{o} \mid \texttt{\_and\_} : \texttt{o} \to \texttt{o} \to \texttt{o} \mid \texttt{\_imp\_} : \texttt{atm} \to \texttt{o} \to \texttt{o} \\ \mid \texttt{all} : (\texttt{expr} \to \texttt{o}) \to \texttt{o} \mid \texttt{ex} : (\texttt{expr} \to \texttt{o}) \to \texttt{o}.\end{aligned}$$

where, although we overloaded the logical connectives of the RL and SL in Section 5, this definition introduces new symbols for the SL connectives so that we may more clearly distinguish them in this section. Hybrid is untyped in the sense that all object-level data have type `expr`. It provides a set of operators on this type that is built definitionally on the foundation of the meta-language of the underlying theorem prover using a form of de Bruijn indices to represent binding operators; no axioms are introduced (see [16]). We omit most

details and just present the operators and some predicates about them used in this paper. The operators with their types are:

$$\begin{array}{ll} \texttt{CON} : \texttt{con} \to \texttt{expr} & \texttt{APP} : \texttt{expr} \to \texttt{expr} \to \texttt{expr} \\ \texttt{VAR} : \texttt{var} \to \texttt{expr} & \texttt{LAM} : (\texttt{expr} \to \texttt{expr}) \to \texttt{expr} \end{array}$$

where the first three come directly from the inductive definition of `expr`. The type `var` is defined to be the natural numbers and can be used to represent free variables, and `con` will be defined later to represent the constants of an OL.

The inductive definition is completed by two operators for de Bruijn representation, `BND` similar to `VAR` used for bound variables, and `ABS` : `expr` → `expr`. The Hybrid library includes a series of definitions used to define `LAM` from the primitive operators (see Section 2.1 of [16]). `LAM` provides the capability to express OL syntax using HOAS. Expanding its definition fully down to primitives gives the low-level de Bruijn representation, which is hidden from the user when reasoning about meta-theory. In fact, `BND`, `ABS`, and the definition of `LAM` will not appear subsequently in this paper. Only the constants (type `con`) together with `APP` and `LAM` will be used to define operators for OL syntax, such as `app` and `lam` of the untyped lambda-calculus. Once defined, such definitions never need to be expanded by the user. Thus, even `APP` and `LAM` will appear only in the definitions and never in proofs.

Two other predicates from the Hybrid library will appear in the proof development, `proper` : `expr` → `Prop` and `abstr` : (`expr` → `expr`) → `Prop`. The `proper` predicate rules out terms that have occurrences of bound variables that do not have a corresponding binder (*dangling indices*).[11] The `abstr` predicate is applied to arguments of `LAM` and rules out meta-level functions that do not encode object-level syntax.

Figure 12 is mapped fairly directly to a Coq inductive definition of type `list atm` → `nat` → `o` → `Prop`, in most cases by simply replacing the connectives shown in the figure with those in the definition of type `o`. The only clause that differs is the one for generic judgments, which in Hybrid is:

$$\texttt{s\_all}: \quad (\forall x.\, \texttt{proper}\, x \to \{\Gamma \vdash_n G\, x\}) \to \{\Gamma \vdash_{n+1} \texttt{all}\ (\lambda x.G\, x)\}.$$

Note the use of the universal quantifier of the RL. By using this quantifier, we do not distinguish variables occurring in a context from terms. The consequences of this choice will be discussed later. Note also the `proper` annotation that is required here. In addition, explicit proof heights are included in sequents and many proofs will be by complete induction on the height of a proof. In this section $\{\Gamma \vdash G\}$ is an abbreviation for $\exists n.\, \{\Gamma \vdash_n G\}$. We distinguish sequents of the SL from those of the RL by writing the latter as:

$$\frac{\mathtt{h}_1 : A_1 \quad \mathtt{h}_2 : A_2 \quad \cdots \quad \mathtt{h}_n : A_n}{A}$$

where $A_1, \ldots, A_n, A$ are Coq formulas (of type `Prop`) and $\mathtt{h}_1, \ldots, \mathtt{h}_n$ are hypothesis names. (The latter are sometimes omitted.) Such sequents appear in the description of formal proofs, which are often described in a goal-directed manner. When there is more than one subgoal with the same set of hypotheses, we write more than one conclusion below the line and number them.

[11] Hybrid 0.2 described in [32], which is implemented in Isabelle/HOL, includes an improvement that doesn't require the `proper` predicate, but this improvement has not yet been ported to the Coq version used in this paper.

$$
\begin{array}{lll}
\texttt{Inductive prog} : \texttt{atm} \to \texttt{o} \to \texttt{Prop} := & & \\
\texttt{tm\_a} : & & \texttt{prog}\ (\texttt{is\_tm}\ (\texttt{app}\ M\ N))\ (\langle \texttt{is\_tm}\ M\rangle\ \texttt{and}\ \langle \texttt{is\_tm}\ N\rangle) \\
\texttt{tm\_l} : & \texttt{abstr}\, M \to & \texttt{prog}\ (\texttt{is\_tm}\ (\texttt{lam}\ M)) \\
 & & \quad \texttt{all}\ \lambda x.\ (\texttt{is\_tm}\ x)\ \texttt{imp}\ \langle \texttt{is\_tm}\ (Mx)\rangle \\
\texttt{ae\_a} : & & \texttt{prog}\ (\texttt{aeq}\ (\texttt{app}\ M_1\ M_2)\ (\texttt{app}\ N_1\ N_2)) \\
 & & \quad \langle \texttt{aeq}\ M_1\ N_1\rangle\ \texttt{and}\ \langle \texttt{aeq}\ M_2\ N_2\rangle \\
\texttt{ae\_l} : \texttt{abstr}\, M \to \texttt{abstr}\, N \to & & \texttt{prog}\ (\texttt{aeq}\ (\texttt{lam}\ M)\ (\texttt{lam}\ N)) \\
 & & \quad \texttt{all}\ \lambda x.\ (\texttt{aeq}\ x\ x)\ \texttt{imp}\ \langle \texttt{aeq}\ (Mx)\ (Nx)\rangle.
\end{array}
$$

Fig. 13 Object-language inference rules in Hybrid

To illustrate the Coq encoding of OL inference rules, we present the syntax of the untyped lambda-calculus and `prog` clauses for algorithmic equality (`ae_e` and `ae_l`) as presented in Section 2.1. We fill in the definition of `con`, define operators `app` and `lam` using the operators defined earlier for `expr`, and fill in the definition of `atm`, which includes `aeq` as well as the `is_tm` predicate for well-formedness of terms.

$$
\begin{array}{l}
\texttt{Inductive con} := \texttt{cAPP} : \texttt{con} \mid \texttt{cLAM} : \texttt{con}. \\
\texttt{Definition app} := \lambda M, N.\ \texttt{APP}\ (\texttt{APP}\ (\texttt{CON cAPP})\ M)\ N. \\
\texttt{Definition lam} := \lambda M.\ \texttt{APP}\ (\texttt{CON cLAM})\ (\texttt{LAM}\ \lambda x. M\ x). \\
\texttt{Inductive atm} := \texttt{is\_tm} : \texttt{expr} \to \texttt{atm} \mid \texttt{aeq} : \texttt{expr} \to \texttt{expr} \to \texttt{atm}.
\end{array}
$$

The inference rules are defined as the inductive definition of `prog` in Fig. 13. Both the definition of `prog` and the definitions above are automatically generated by the tool described in [23], which translates ORBI files to Coq script. This script imports the Hybrid libraries and sets up the basic definitions needed to reason about a particular OL, including definitions for context schemas and relations.

Two reasons for including the `prog` clauses `tm_l` and `tm_a` were discussed in the previous section. First, they provide induction on well-formedness derivations, which is often useful, especially since induction directly on terms is not currently available.[12] Second, they are important for proving internal adequacy. The latter is especially important in Hybrid, since as mentioned above, Hybrid expressions are untyped. The `abstr` conditions are also important for this purpose. An example is discussed in Section 6.3.3; more details on both external and internal adequacy can be found in [16].

The `prog` clauses for all other judgments considered here are encoded via translation from the specifications in the `Rules` sections of the ORBI files (see Appendix A) in a systematic manner. The only non-systematic aspect of the encoding of inference rules is the insertion of well-formedness annotations, such as those in the `de_l` and `de_r` clauses on page 27. (The same ones are used in both Hybrid and Abella.) We add special directives to the ORBI files (which are not shown in the appendix, but are discussed in the companion paper) that indicate which annotations to include.

For all benchmark problems that we are able to formalize in Hybrid, we do both the R and G versions.

[12] Please see [2] and [7] for work in this direction.

6.1 Basic Linear Context Extension

6.1.1 R Version

We begin by defining the context relation `xaR` (see the `Definitions` section of Appendix A.1) in Coq:

$$\begin{aligned}&\texttt{Inductive xaR} : \texttt{list atm} \to \texttt{list atm} \to \texttt{Prop} :=\\ &\qquad\qquad\qquad\qquad \texttt{xaR nil nil}\\ &\texttt{xaR}\ \Phi_x\ \Phi_a \to \texttt{proper}\, x \to \texttt{xaR}\ (\texttt{is\_tm}\, x :: \Phi_x)\ (\texttt{aeq}\, x\, x :: \Phi_a).\end{aligned}$$

Recall our convention of leaving out explicit quantifiers at the top-level, which here includes universal quantification over x as well as Φ_x and Φ_a in the second clause of the definition. Here x is an arbitrary term, since as mentioned, we do not distinguish variables occurring in a context from terms. Note that whenever the typing declaration `x:tm` appears in a context in an ORBI specification, the corresponding Hybrid definition contains the atomic formula `is_tm` x. (Similarly, in the next subsection, declarations of the form `a:tp` in ORBI will appear as formulas in Hybrid.) Also note that `aeq` x x appears by itself instead of in a block with `is_tm` x. As stated in the previous section, we leave out well-formedness annotations from `prog` clauses and context relations whenever possible. Finally, note also the presence of the required `proper` annotation.

We start with the Hybrid version of membership Lemma 6 from Section 2.2.

H-Lemma 1 (Context Membership)

$\forall M, \Phi_x, \Phi_a.\ \texttt{xaR}\ \Phi_x\ \Phi_a\ \to\ \texttt{is\_tm}\, M \in \Phi_x\ \to\ \texttt{aeq}\, M\, M \in \Phi_a.$

The proof is by induction on the first premise, and is simple and follows the same pattern as all other context membership lemmas. Current work on Hybrid includes writing a Coq tactic to automate these proofs, which will be straightforward. Note that the second implication in the statement of this lemma could be replaced by equivalence, since it certainly holds in both directions. We only state such lemmas using implication in the direction(s) that are needed, so that they can be used in automating proofs via `eauto`. The `eauto` tactic is parametrized by a set of lemmas marked as hints and we generally add context membership, strengthening, and weakening lemmas as hints, which together provide significant automation of proofs.

We give the proof of the first benchmark in some detail.

H-Theorem 2 (Admissibility of Reflexivity, R Version)

$\texttt{xaR}\ \Phi_x\ \Phi_a \to \{\Phi_x \vdash_n \langle \texttt{is\_tm}\, M\rangle\} \to \{\Phi_a \vdash_n \langle \texttt{aeq}\, M\, M\rangle\}.$

Proof The proof is by complete induction on n with induction hypothesis $IH := \forall i, M, \Phi_x, \Phi_a.\ i < n \to \texttt{xaR}\ \Phi_x\ \Phi_a \to \{\Phi_x \vdash_i \langle \texttt{is\_tm}\ M\rangle\} \to \{\Phi_a \vdash_i \langle \texttt{aeq}\ M\ M\rangle\}$.

A derivation of $\{\Phi_x \vdash_n \langle \texttt{is\_tm}\ M\rangle\}$ must end in an application of the last two clauses of the definition of the SL (`s_init` or `s_bc` in Fig. 12 on page 26). In general, the `s_init` case in Hybrid proofs corresponds to the variable case in the informal proof. Here, we know that $(\texttt{is\_tm}\ M) \in \Phi_x$. By H-Lemma 1, $(\texttt{aeq}\ M\ M) \in \Phi_a$. We use this fact and simply apply `s_init` to obtain $\{\Phi_a \vdash_n \langle \texttt{aeq}\ M\ M\rangle\}$, as desired.

All other cases in the informal proofs are covered by the s_bc case in Hybrid proofs. Here, when the derivation ends in s_bc, it must be the case that one of the two clauses defining the is_tm judgment (tm_a or tm_l in Fig. 13 on page 30) was used.

Case tm_a: We know that M has the form (app M_1 M_2), and we must show:

$$\frac{\mathtt{h}_1 : IH \quad \mathtt{h}_2 : \mathtt{xaR}\ \Phi_x\ \Phi_a \quad \mathtt{h}_3 : \{\Phi_x \vdash_n \langle \mathtt{is\_tm}\ (\mathtt{app}\ M_1\ M_2)\rangle\}}{\{\Phi_a \vdash_n \langle \mathtt{aeq}\ (\mathtt{app}\ M_1\ M_2)\ (\mathtt{app}\ M_1\ M_2)\rangle\}}$$

By repeated inversion of the SL rules on $\mathtt{h}_3$, and repeated backward application of these rules to the conclusion, the above goal reduces to the following 2 subgoals:

$$\frac{IH \quad \mathtt{xaR}\ \Phi_x\ \Phi_a \quad \{\Phi_x \vdash_{n-2} \langle \mathtt{is\_tm}\ M_1\rangle\} \quad \{\Phi_x \vdash_{n-2} \langle \mathtt{is\_tm}\ M_2\rangle\}}{}$$

1. $\{\Phi_a \vdash_{n-2} \langle \mathtt{aeq}\ M_1\ M_1\rangle\}$
2. $\{\Phi_a \vdash_{n-2} \langle \mathtt{aeq}\ M_2\ M_2\rangle\}$

which are both provable by the induction hypothesis.

Case tm_l: We know that M has the form (lam M') and we must show:

$$\frac{\mathtt{h}_1 : IH \quad \mathtt{h}_2 : \mathtt{xaR}\ \Phi_x\ \Phi_a \quad \mathtt{h}_3 : \{\Phi_x \vdash_n \langle \mathtt{is\_tm}\ (\mathtt{lam}\ M')\rangle\}}{\{\Phi_a \vdash_n \langle \mathtt{aeq}\ (\mathtt{lam}\ M')\ (\mathtt{lam}\ M')\rangle\}}$$

Again, by repeated inversion of the SL rules on $\mathtt{h}_3$, and repeated backward application of these rules to the conclusion, the above goal reduces to the following subgoal:

$$\frac{IH \quad \mathtt{xaR}\ \Phi_x\ \Phi_a \quad \mathtt{proper}\ x \quad \{\mathtt{is\_tm}\ x :: \Phi_x \vdash_{n-3} \langle \mathtt{is\_tm}\ (M'x)\rangle\}}{\{\mathtt{aeq}\ x\ x :: \Phi_a \vdash_{n-3} \langle \mathtt{aeq}\ (M'x)\ (M'x)\rangle\}}$$

By definition of xaR, we can conclude xaR (is_tm x :: Φ_x) (aeq x x :: Φ_a), and then apply the induction hypothesis to complete the proof. □

6.1.2 G Version

We begin by defining the generalized context inductively.

$$\begin{aligned}\mathtt{Inductive\ xaG} : \mathtt{list\ atm} \to \mathtt{Prop} := \quad & \mathtt{xaG\ nil} \\ | \ \mathtt{xaG}\ \Phi_{xa} \to \mathtt{proper}\ x \to & \mathtt{xaG}\ (\mathtt{is\_tm}\ x :: \mathtt{aeq}\ x\ x :: \Phi_{xa}).\end{aligned}$$

We take this opportunity to remark that the block structure of a context as introduced in Section 2 is enforced by pattern matching on the cons notation (::), which in this case requires the list to either be empty or to always contain the declaration is_tm x followed by aeq x x.[13]

In order to formalize the lambda case of Theorem 8 we need a *d-str* lemma. Here, we state it as a corollary (H-Corollary 4 below) of the more general H-Lemma 3. (Although strengthening was not needed in the informal proof, it is required here because of the fact that we leave out assumptions of the form is_tm x when defining the inference rules for aeq. See Fig. 13 on page 30).

[13] We omit the Coq inductive definitions for contexts in the rest of this section. G versions like this one are obtained directly from the corresponding definition in the Schemas section of the ORBI files in the appendix. Similarly, inductive definitions of context relations such as xaR are obtained from the corresponding Definitions sections.

H-Lemma 3

$(\forall x, y.\mathtt{aeq}\, x\, y \in \Gamma_1 \longleftrightarrow \mathtt{aeq}\, x\, y \in \Gamma_2) \rightarrow \{\Gamma_1 \vdash_n \langle \mathtt{aeq}\, M\, N\rangle\} \rightarrow \{\Gamma_2 \vdash_n \langle \mathtt{aeq}\, M\, N\rangle\}$.

Note that this lemma preserves the height n. It is proved by a straightforward induction on proof height n. We state it in this general form because other instances of both *d-str* and *c-str* required later also follow directly from it.

H-Corollary 4 (D-Strengthening)

$\{\mathtt{is\_tm}\, x :: \mathtt{aeq}\, x\, x :: \Gamma \vdash_n \langle \mathtt{aeq}\, M\, N\rangle\} \rightarrow \{\mathtt{aeq}\, x\, x :: \Gamma \vdash_n \langle \mathtt{aeq}\, M\, N\rangle\}$.

We also need the standard membership lemma (omitted). With these in place, the proof of the admissibility of reflexivity is straightforward. We show it in some detail for comparison.

H-Theorem 5 (Admissibility of Reflexivity, G Version)

$\mathtt{xaG}\ \Phi_{xa} \rightarrow \{\Phi_{xa} \vdash_n \langle \mathtt{is\_tm}\, M\rangle\} \rightarrow \{\Phi_{xa} \vdash_n \langle \mathtt{aeq}\, M\, M\rangle\}$.

Proof Again, the proof is by complete induction on n. The context and application cases are similar to H-Theorem 2, just like these two cases were similar in the informal proofs of Theorems 7 and 8 presented in the companion paper [17]. In the abstraction case (`tm_l`), we know that M has the form $(\mathtt{lam}\, M')$ and we must show:

$$\frac{\mathtt{h}_1 : IH \quad \mathtt{h}_2 : \mathtt{xaG}\ \Phi_{xa} \quad \mathtt{h}_3 : \{\Phi_{xa} \vdash_n \langle \mathtt{is\_tm}\, (\mathtt{lam}\, M')\rangle\}}{\{\Phi_{xa} \vdash_n \langle \mathtt{aeq}\, (\mathtt{lam}\, M')\ (\mathtt{lam}\, M')\rangle\}}$$

By repeated inversion of the SL rules on $\mathtt{h}_3$, and repeated backward application of these rules to the conclusion, the above goal reduces to the following subgoal:

$$\frac{IH \quad \mathtt{xaG}\ \Phi_{xa} \quad \mathtt{h}_4 : \mathtt{proper}\, x \quad \mathtt{h}_5 : \{\mathtt{is\_tm}\, x :: \Phi_{xa} \vdash_{n-3} \langle \mathtt{is\_tm}\, (M' x)\rangle\}}{\{\mathtt{aeq}\, x\ x :: \Phi_{xa} \vdash_{n-3} \langle \mathtt{aeq}\, (M' x)\ (M' x)\rangle\}}$$

We apply *d-wk* to $\mathtt{h}_5$ via SL-Theorem 1 (b) to obtain:

$$\{\mathtt{is\_tm}\, x :: \mathtt{aeq}\, x\ x :: \Phi_{xa} \vdash_{n-3} \langle \mathtt{is\_tm}\, (M' x)\rangle\}$$

By definition of `xaG`, we can conclude $\mathtt{xaG}\ (\mathtt{is\_tm}\ x :: \mathtt{aeq}\ x\ x :: \Phi_{xa})$, and then apply *IH* to obtain $\{\mathtt{is\_tm}\, x :: \mathtt{aeq}\, x\ x :: \Phi_{xa} \vdash_{n-3} \langle \mathtt{aeq}\, (M' x)\ (M' x)\rangle\}$. Finally, we apply *d-str* via H-Corollary 4 to obtain $\{\mathtt{aeq}\, x\ x :: \Phi_{xa} \vdash_{n-3} \langle \mathtt{aeq}\, (M' x)\ (M' x)\rangle\}$ as desired. □

For admissibility of reflexivity, which is our simplest benchmark, there is very little difference between the R and G versions in Hybrid. As we get to more complicated examples, the G versions require more (sometimes many more) strengthening, weakening, and promotion lemmas as compared to the R versions, but other than that, present no further complications.

The proofs of admissibility of symmetry and transitivity for algorithmic equality use the following membership lemma.

H-Lemma 6 (Context Inversion) $\mathtt{aG}\ \Phi_a \rightarrow \mathtt{aeq}\, M\, N \in \Phi_a \rightarrow M = N$.

Here the inductive definition of aG is obtained from the definition of schema xaG in Appendix A.1, in this case omitting the well-formed term annotations, i.e., in (aG Φ_a), the context Φ_a contains only atoms of the form (aeq x x). Like reflexivity, the proofs of symmetry and transitivity are straightforward and are stated as follows in Hybrid.

H-Theorem 7 (Symmetry and Transitivity)
1. aG $\Phi_a \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ M\ N\rangle\} \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ N\ M\rangle\}$.
2. aG $\Phi_a \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ M\ L\rangle\} \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ L\ N\rangle\} \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ M\ N\rangle\}$.

We note here that the abstraction cases of some formal proofs require explicit reasoning about Hybrid's extensional equality [16]. The only two examples in this paper where such details appear are the lam cases of H-Theorem 7 (2) above and the failed proof of H-Attempt 35 in Section 6.6.

6.2 Linear Context Extensions with Alternative Declarations

The next benchmark extends reflexivity of equality to the polymorphic lambda-calculus. We first extend the definitions on page 29 to include new constants and operators tapp and tlam and replace the definition of atm with one that includes four predicates: is_tp, is_tm, atp, and aeq, thus implementing the Syntax and Judgments sections of the ORBI file in Appendix A.2. The definition of prog in Fig. 13 is extended to include all the declarations in the Rules section, plus clauses for well-formedness of tp.

6.2.1 G Version

The inductive definitions for context schemas atpG and aeqG implement directly the block structure in the corresponding schema declarations in Appendix A.2, where the former includes blocks of the form (is_tp α :: atp α α) and the latter includes, in addition, blocks of the form (is_tm x :: aeq x x). The statement and proof of admissibility of reflexivity for types is similar to the case for terms in the untyped lambda-calculus. We simply state the required lemmas and theorem. The lemmas include the usual membership lemma as well as a *d-str* lemma and corollary (similar to H-Lemma 3 and H-Corollary 4).

H-Lemma 8 (Context Membership)

atpG $\Phi_{atp} \rightarrow$ is_tp $A \in \Phi_{atp} \rightarrow$ atp $A\ A \in \Phi_{atp}$.

H-Lemma 9

$(\forall \alpha, \alpha'.\texttt{atp}\ \alpha\ \alpha' \in \Gamma_1 \longleftrightarrow \texttt{atp}\ \alpha\ \alpha' \in \Gamma_2) \rightarrow \{\Gamma_1 \vdash_n \langle \texttt{atp}\ A\ B\rangle\} \rightarrow \{\Gamma_2 \vdash_n \langle \texttt{atp}\ A\ B\rangle\}$.

H-Corollary 10 (D-Strengthening)

$\{\texttt{is\_tp}\ \alpha :: \texttt{atp}\ \alpha\ \alpha :: \Gamma \vdash_n \langle \texttt{atp}\ A\ B\rangle\} \rightarrow \{\texttt{atp}\ \alpha\ \alpha :: \Gamma \vdash_n \langle \texttt{atp}\ A\ B\rangle\}$.

H-Theorem 11 (Admissibility of Reflexivity for Types, G Version)

atpG $\Phi_{atp} \rightarrow \{\Phi_{atp} \vdash_n \langle \texttt{is\_tp}\ A\rangle\} \rightarrow \{\Phi_{atp} \vdash_n \langle \texttt{atp}\ A\ A\rangle\}$.

Proving reflexivity of equality for terms of the polymorphic lambda-calculus requires promotion of the corresponding lemma for types (Lemma 12), whose proof requires *c-str* and *c-wk*. These rules are expressed in [17, Section 2.3] using explicit strengthening functions (see Property 4). The particular function used here is $\mathsf{rm}^*_{\mathsf{is\_tm}\ x;\mathsf{aeq}\ x\ x}$ and in Hybrid we define it directly as a Coq recursive function.

$$
\begin{array}{l}
\texttt{Fixpoint rm\_aeq2atp} : \texttt{atm}\ \mathit{list} \to \texttt{atm}\ \mathit{list} := \\
\quad \texttt{rm\_aeq2atp}\ (\texttt{is\_tp}\ \alpha :: \texttt{atp}\ \alpha'\ \alpha'' :: \Phi_{aeq}) = \\
\qquad\qquad\qquad (\texttt{is\_tp}\ \alpha :: \texttt{atp}\ \alpha'\ \alpha'' :: \texttt{rm\_aeq2atp}\ \Phi_{aeq}) \\
\quad \texttt{rm\_aeq2atp}\ (\texttt{is\_tm}\ \_ :: \texttt{aeq}\ \_\ \_ :: \Phi_{aeq}) \quad = (\texttt{rm\_aeq2atp}\ \Phi_{aeq}) \\
\quad \texttt{rm\_aeq2atp}\ \_ \quad = \texttt{nil}.
\end{array}
$$

Note that this strengthening function transforms a context having schema `aeqG` (see Appendix A.2) to one having schema `atpG` by removing an entire alternative and leaving the other intact. The third clause of this definition is required since this function can be called with any (`atm` *list*), though this clause is never used since the function is only called on contexts having schema `aeqG`. This restriction is satisfied, for example, in the statement of the following lemma, which directly relates contexts satisfying these two schemas.

H-Lemma 12 $\texttt{aeqG}\ \Phi_{aeq} \to \texttt{atpG}\ (\texttt{rm\_aeq2atp}\ \Phi_{aeq})$.

This result was left implicit in the informal proof. It is proved by a simple induction on the definition of `aeqG`. We then prove the required *c-str* and *c-wk* lemmas.

H-Lemma 13

1. $(\forall \alpha.\texttt{is\_tp}\ \alpha \in \Gamma_1 \longleftrightarrow \texttt{is\_tp}\ \alpha \in \Gamma_2) \to \{\Gamma_1 \vdash_n \langle \texttt{is\_tp}\ A\rangle\} \to \{\Gamma_2 \vdash_n \langle \texttt{is\_tp}\ A\rangle\}$.
2. $\texttt{aeqG}\ \Phi_{aeq} \to (\forall \alpha.\texttt{is\_tp}\ \alpha \in \Phi_{aeq} \longleftrightarrow \texttt{is\_tp}\ \alpha \in (\texttt{rm\_aeq2atp}\ \Phi_{aeq}))$.
3. $\texttt{aeqG}\ \Phi_{aeq} \to (\forall \alpha, \alpha'.\texttt{atp}\ \alpha\ \alpha' \in \Phi_{aeq} \longleftrightarrow \texttt{atp}\ \alpha\ \alpha' \in (\texttt{rm\_aeq2atp}\ \Phi_{aeq}))$.

Proof Note that (1) is similar to H-Lemma 9. (2) and (3) follow from a simple induction and the definition of `rm_aeq2atp`. □

H-Corollary 14 (C-Strengthening/Weakening)

1. $\texttt{aeqG}\ \Phi_{aeq} \to \{\Phi_{aeq} \vdash_n \langle \texttt{is\_tp}\ A\rangle\} \to \{\texttt{rm\_aeq2atp}\ \Phi_{aeq} \vdash_n \langle \texttt{is\_tp}\ A\rangle\}$
2. $\texttt{aeqG}\ \Phi_{aeq} \to \{\texttt{rm\_aeq2atp}\ \Phi_{aeq} \vdash_n \langle \texttt{atp}\ A\ B\rangle\} \to \{\Phi_{aeq} \vdash_n \langle \texttt{atp}\ A\ B\rangle\}$

Proof (1) is a corollary of H-Lemma 13 (1) and (2), and (2) is a corollary of H-Lemma 9 and H-Lemma 13 (3). □

Now, we can state and prove the required promotion lemma. We give the proof explicitly, though the formalized proof is fully automatic once the above lemmas are added to the hint database.

H-Lemma 15 (**Promotion**)
$\texttt{aeqG}\ \Phi_{aeq} \to \{\Phi_{aeq} \vdash_n \langle \texttt{is\_tp}\ A\rangle\} \to \{\Phi_{aeq} \vdash_n \langle \texttt{atp}\ A\ A\rangle\}$.

Proof

$\texttt{atpG}\ (\texttt{rm\_aeq2atp}\ \Phi_{aeq})$	by H-Lemma 12
$\{\texttt{rm\_aeq2atp}\ \Phi_{aeq} \vdash_n \langle \texttt{is\_tp}\ A \rangle\}$	by *c-str*: H-Corollary 14 (1)
$\{\texttt{rm\_aeq2atp}\ \Phi_{aeq} \vdash_n \langle \texttt{atp}\ A\ A \rangle\}$	H-Theorem 11
$\{\Phi_{aeq} \vdash_n \langle \texttt{atp}\ A\ A \rangle\}$	by *c-wk*: H-Corollary 14 (2)

□

Now, turning to the proof of admissibility of reflexivity for terms, in addition to promotion, we also need 2 membership lemmas and 2 *d-str* lemmas. We omit them here. The latter are needed in the `ae_l` and `ae_tl` cases, and are similar to H-Corollary 4 and H-Corollary 10.

H-Theorem 16 (**Admissibility of Reflexivity for Terms, G Version**)

$\texttt{aeqG}\ \Phi_{aeq} \rightarrow \{\Phi_{aeq} \vdash_n \langle \texttt{is\_tm}\ M \rangle\} \rightarrow \{\Phi_{aeq} \vdash_n \langle \texttt{aeq}\ M\ M \rangle\}$.

Proof As in the informal proof in the companion paper of Theorem 13, we show only the case for application of terms to types.

Case `ae_ta`: We know that M has the form $(\texttt{tapp}\ M'A)$ and we must show:

$$\frac{\texttt{h}_1 : IH \quad \texttt{h}_2 : \texttt{aeqG}\ \Phi_{aeq} \quad \texttt{h}_3 : \{\Phi_{aeq} \vdash_n \langle \texttt{is\_tm}\ (\texttt{tapp}\ M'\ A) \rangle\}}{\{\Phi_{aeq} \vdash_n \langle \texttt{aeq}\ (\texttt{tapp}\ M'\ A)\ (\texttt{tapp}\ M'\ A) \rangle\}}$$

By repeated inversion of the SL rules on $\texttt{h}_3$, and repeated backward application of these rules to the conclusion, the above goal reduces to the following 2 subgoals:

$$\frac{IH \quad \texttt{aeqG}\ \Phi_{aeq} \quad \{\Phi_{aeq} \vdash_{n-2} \langle \texttt{is\_tm}\ M' \rangle\} \quad \{\Phi_{aeq} \vdash_{n-2} \langle \texttt{is\_tp}\ A \rangle\}}{}$$

1. $\{\Phi_{aeq} \vdash_{n-2} \langle \texttt{aeq}\ M'\ M' \rangle\}$
2. $\{\Phi_{aeq} \vdash_{n-2} \langle \texttt{atp}\ A\ A \rangle\}$

The first is provable by *IH* and the second by promotion (H-Lemma 15). □

6.2.2 R Version

The context relations $(\texttt{atpR}\ \ \Phi_\alpha\ \Phi_{atp})$ and $(\texttt{aeqR}\ \ \Phi_{\alpha x}\ \Phi_{aeq})$ are defined as Coq inductive predicates implementing `atpR` and `aeqR` from the `Definitions` section of Appendix A.2, similar to the inductive definition in Section 6.1.1 implementing `xaR` from Appendix A.1. Just as in the Hybrid definition of `xaR`, we remove the term and type well-formedness annotations from the second argument to the relations.

As before, *d-str* lemmas are not needed in the R version of admissibility of reflexivity, and membership lemmas are similar to H-Lemma 1. Admissibility for types is stated below with proof omitted; it is similar to those already shown.

H-Theorem 17 (**Admissibility of Reflexivity for Types, R Version**)

$\texttt{atpR}\ \Phi_\alpha\ \Phi_{atp} \rightarrow \{\Phi_\alpha \vdash_n \langle \texttt{is\_tp}\ A \rangle\} \rightarrow \{\Phi_{atp} \vdash_n \langle \texttt{atp}\ A\ A \rangle\}$.

In order to implement relational strengthening Lemma 15 (see Section 2.2) in Hybrid, we witness Φ_α and Φ_{atp} directly by applying strengthening functions to $\Phi_{\alpha x}$ and Φ_{aeq}, respectively. These functions are implemented similarly to `rm_aeq2atp` in the G version (Section 6.2.1). We only show the main clauses of the definitions.

$$\begin{aligned}
&\texttt{rm_alphx2alph}\ (\texttt{is_tp}\ \alpha :: \Phi_{\alpha x}) &&= (\texttt{is_tp}\ \alpha :: \texttt{rm_alphx2alph}\ \Phi_{\alpha x})\\
&\texttt{rm_alphx2alph}\ (\texttt{is_tm}\ _ :: \Phi_{\alpha x}) &&= (\texttt{rm_alphx2alph}\ \Phi_{\alpha x})\\
&\texttt{rm_aeq2atp}\ (\texttt{atp}\ \alpha'\ \alpha'' :: \Phi_{aeq}) &&= (\texttt{atp}\ \alpha'\ \alpha'' :: \texttt{rm_aeq2atp}\ \Phi_{aeq})\\
&\texttt{rm_aeq2atp}\ (\texttt{aeq}\ _\ _ :: \Phi_{aeq}) &&= (\texttt{rm_aeq2atp}\ \Phi_{aeq})
\end{aligned}$$

The former transforms a context having schema `axG` to one having schema `aG`, and the latter transforms a context having schema `aeqG` to one having schema `atpG`. Using these functions, relational strengthening is stated as follows.

H-Lemma 18 (**Relational Strengthening**)

$\texttt{aeqR}\ \Phi_{\alpha x}\ \Phi_{aeq} \to \texttt{atpR}\ (\texttt{rm\_alphx2alph}\ \Phi_{\alpha x})\ (\texttt{rm\_aeq2atp}\ \Phi_{aeq})$.

Note that this lemma is similar to H-Lemma 12 of the G version. We next state the required *c-str* and *c-wk* lemmas.

H-Corollary 19 (**C-Strengthening/Weakening**)

1. $\texttt{aeqR}\ \Phi_{\alpha x}\ \Phi_{aeq} \to \{\Phi_{\alpha x} \vdash_n \langle \texttt{is\_tp}\ A\rangle\} \to \{\texttt{rm\_alphx2alph}\ \Phi_{\alpha x} \vdash_n \langle \texttt{is\_tp}\ A\rangle\}$.
2. $\texttt{aeqR}\ \Phi_{\alpha x}\ \Phi_{aeq} \to \{\texttt{rm\_aeq2atp}\ \Phi_{aeq} \vdash_n \langle \texttt{atp}\ A\ B\rangle\} \to \{\Phi_{aeq} \vdash_n \langle \texttt{atp}\ A\ B\rangle\}$.

This result is stated as a corollary since it follows from the same kinds of lemmas (omitted here) as H-Corollary 14. The following promotion lemma follows from the above lemmas in a similar series of steps to the proof of Lemma 15.

H-Lemma 20 (**Promotion**)

$\texttt{aeqR}\ \Phi_{\alpha x}\ \Phi_{aeq} \to \{\Phi_{\alpha x} \vdash_n \langle \texttt{is\_tp}\ A\rangle\} \to \{\Phi_{aeq} \vdash_n \langle \texttt{atp}\ A\ A\rangle\}$.

Unlike H-Lemma 15, this proof cannot be fully automated in Coq using `eauto` because there are too many existential variables to fill in. In general, there are fewer when a single generalized context is used.

We now state the final result, admissibility of reflexivity for terms, whose proof is similar to the proof of H-Theorem 16, except there is no need for *d-str*.

H-Theorem 21 (**Admissibility of Reflexivity for Terms, R Version**)

$\texttt{aeqR}\ \Phi_{\alpha x}\ \Phi_{aeq} \to \{\Phi_{\alpha x} \vdash_n \langle \texttt{is\_tm}\ M\rangle\} \to \{\Phi_{aeq} \vdash_n \langle \texttt{aeq}\ M\ M\rangle\}$.

In this subsection, we have shown both the G and R versions in detail, to make explicit all the required lemmas (e.g., *c-* and *d-strengthening* and *weakening*, and promotion) and to illustrate that Hybrid is flexible enough to do different versions of this proof. In doing so, we follow the informal proofs very closely. If the goal were to just complete any proof and move on, we would have chosen the R version, since it is slightly simpler. In fact, an

even simpler R version is done in Abella in Section 7.2, and that proof is also possible, and perhaps preferable in Hybrid.[14]

6.3 Non-linear Context Extensions

For the completeness of equality benchmark, the definitions of the Hybrid constants include `app` and `lam` and the definition of `atm` includes predicates `is_tm`, `aeq`, and `deq`. (See page 29 and Appendix A.1.) As mentioned in Section 5, the encoding of the inference rules for `deq` require some well-formedness annotations in order to prove adequacy. We briefly discuss the statements and proofs of internal adequacy at the end of this subsection (Section 6.3.3) after illustrating the R version (Section 6.3.1) and the G version (Section 6.3.2) of the completeness theorem. As usual, the G version of the proof is more complicated than the R version.

6.3.1 R Version

The Hybrid definition of $(\texttt{daR}\ \Phi_a\ \Phi_{xd})$ implements `daR` in Appendix A.1. The first argument Φ_a does not contain the term well-formedness annotation, but Φ_{xd} does. We once again formalize relational strengthening (see H-Lemma 18) using a strengthening function `rm_xd2x` , whose main clause is:

$$\texttt{rm\_xd2x}\ (\texttt{is\_tm}\ x :: \texttt{deq}\ \_\ \_ :: \Phi_{xd}) = (\texttt{is\_tm}\ x :: \texttt{rm\_xd2x}\ \Phi_{xd})$$

Unlike in Section 6.2 where all the strengthening functions removed an alternative from a context schema, this one relates two schemas each with just one alternative and modifies every block by removing one of the two atoms.

H-Lemma 22 (Relational Strengthening)

1. $\texttt{daR}\ \Phi_a\ \Phi_{xd} \rightarrow \texttt{xaR}\ (\texttt{rm\_xd2x}\ \Phi_{xd})\ \Phi_a$.
2. $\texttt{daR}\ \Phi_a\ \Phi_{xd} \rightarrow \texttt{aG}\ \Phi_a$.

In the above lemma, the first statement is the Hybrid version of Lemma 20 in Section 2.2, needed to prove promotion for reflexivity (Lemma 21). The promotion lemmas for symmetry and transitivity have much simpler statements and proofs, and were omitted from the informal proof. Here we make them explicit. Their proofs require the second statement above, but other than that are simple.

Promotion for reflexivity requires one *c-str* lemma, as follows.

H-Lemma 23 (C-Strengthening)

$$\texttt{daR}\ \Phi_a\ \Phi_{xd} \rightarrow \{\Phi_{xd} \vdash_n \langle \texttt{is\_tm}\ A\rangle\} \rightarrow \{\texttt{rm\_xd2x}\ \Phi_{xd} \vdash_n \langle \texttt{is\_tm}\ A\rangle\}.$$

[14] The version presented there (A-Theorem 11) illustrates the trade-off between choosing the smallest context relation possible for each lemma (as is done here), and choosing one that leads to a simpler but less modular proof development. There are also other proof developments described in Section 7 that diverge from the informal versions in the companion paper, which would also present no problems in Hybrid (e.g., A-Theorem 18).

H-Lemma 24 (Promotion)

1. $\texttt{daR}\ \Phi_a\ \Phi_{xd} \rightarrow \{\Phi_{xd} \vdash_n \langle \texttt{is\_tm}\ M\rangle\} \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ M\ M\rangle\}$.
2. $\texttt{daR}\ \Phi_a\ \Phi_{xd} \rightarrow \{\Phi_{xd} \vdash_n \langle \texttt{aeq}\ M\ N\rangle\} \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ N\ M\rangle\}$.
3. $\texttt{daR}\ \Phi_a\ \Phi_{xd} \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ M\ L\rangle\} \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ L\ N\rangle\} \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ M\ N\rangle\}$.

(In contrast, the G version of the above lemma in the next subsection requires several more strengthening and weakening lemmas.)

H-Theorem 25 (Completeness, R Version)

$\texttt{daR}\ \Phi_a\ \Phi_{xd} \rightarrow \{\Phi_{xd} \vdash_n \langle \texttt{deq}\ M\ N\rangle\} \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ M\ M\rangle\}$.

Proof The proof is by induction on n as usual. We discuss the same cases as in the informal proof of Theorem 22.

Case `de_r`: In the case when the last rule in the proof of $\{\Phi_{xd} \vdash_n \langle \texttt{deq}\ M\ N\rangle\}$ is reflexivity, we know that $M = N$ and $\{\Phi_{xd} \vdash_{n-1} \langle \texttt{is\_tm}\ M\rangle\}$. From height weakening (part (a) of SL-Theorem 1), we also have $\{\Phi_{xd} \vdash_n \langle \texttt{is\_tm}\ M\rangle\}$. We can then apply H-Lemma 24 (1) to obtain the desired result.

Case `de_t`: We know that there is an L such that $\{\Phi_{xd} \vdash_{n-1} \langle \texttt{deq}\ M\ L\rangle\}$ and $\{\Phi_{xd} \vdash_{n-1} \langle \texttt{deq}\ L\ N\rangle\}$. By the induction hypothesis, followed by height weakening, we can conclude $\{\Phi_a \vdash_n \langle \texttt{aeq}\ M\ L\rangle\}$ and $\{\Phi_a \vdash_n \langle \texttt{aeq}\ L\ N\rangle\}$. We can then apply H-Lemma 24 (3) to obtain the desired result.

Case `de_l`: This case is similar to other abstraction cases, such as the `tm_l` case of H-Theorem 2 except that height weakening is needed. We know that M and N have the form $(\mathsf{lam}\, x.\, M'x)$ and $(\mathsf{lam}\, x.\, N'x)$, respectively, and we must show:

$$\frac{\texttt{h}_1 : \mathit{IH} \quad \texttt{h}_2 : \texttt{daR}\ \Phi_a\ \Phi_d \quad \texttt{h}_3 : \{\Phi_d \vdash_n \langle \texttt{deq}\ (\mathsf{lam}\, x.\, M'x)\ (\mathsf{lam}\, x.\, N'x)\rangle\}}{\{\Phi_a \vdash_n \langle \texttt{aeq}\ (\mathsf{lam}\, x.\, M'x)\ (\mathsf{lam}\, x.\, N'x)\rangle\}}$$

By repeated inversion of the SL rules on $\texttt{h}_3$, and repeated backward application of these rules to the conclusion, the above goal reduces to the following subgoal:

$$\frac{\mathit{IH} \quad \texttt{daR}\ \Phi_a\ \Phi_d \quad \texttt{h}_4 : \texttt{proper}\ x \qquad \texttt{h}_5 : \{\texttt{is\_tm}\ x :: \texttt{deq}\ x\ x :: \Phi_{da} \vdash_{n-4} \langle \texttt{deq}\ (M'x)\ (N'x)\rangle\}}{\{\texttt{aeq}\ x\ x :: \Phi_{da} \vdash_{n-3} \langle \texttt{aeq}\ (M'x)\ (N'x)\rangle\}}$$

The heights of $\texttt{h}_5$ and the conclusion differ because the `de_l` rule has the additional well-formedness premise and the `ae_l` does not. We simply apply SL-Theorem 1 (a) to obtain $n - 3$ as the height for both and apply *IH*. □

6.3.2 G Version

The definition of $(\texttt{daG}\ \Phi_{da})$ implements as usual the block structure of the corresponding schema declaration in Appendix A.1, which here includes blocks of the form $(\texttt{is\_tm}\ x :: \texttt{deq}\ x\ x :: \texttt{aeq}\ x\ x)$. The contexts `xaG` and `aG` defined in Section 6.1.2 are also used here.

Note that H-Theorem 5 is stated using context Φ_{xa} and that H-Theorem 7 is stated using context Φ_a. Since we will need both theorems here, we need to promote them to Φ_{da}. As in Section 6.2.1, we need strengthening functions and a series of lemmas analogous to H-

Lemmas 12–15. The strengthening functions must strengthen Φ_{da} to Φ_{xa} and Φ_a. The main clauses of these function definitions are:

$$\begin{aligned}
&\texttt{rm_da2xa}\,(\texttt{is_tm}\,z :: \texttt{deq}\,_\,_ :: \texttt{aeq}\,x\,y :: \Phi_{da}) = (\texttt{is_tm}\,z :: \texttt{aeq}\,x\,y :: \texttt{rm_da2xa}\,\Phi_{da})\\
&\texttt{rm_da2a}\,(\texttt{is_tm}\,_ :: \texttt{deq}\,_\,_ :: \texttt{aeq}\,x\,y :: \Phi_{da}) \quad = (\texttt{aeq}\,x\,y :: \texttt{rm_da2a}\,\Phi_{da})
\end{aligned}$$

Unlike in Section 6.2 where all the strengthening functions removed an alternative from a context schema, all those in this subsection involve schemas with just one alternative and modify every block by removing one or more atoms. The lemmas required to prove promotion are as follows. (The strengthening and weakening lemmas are again stated as a corollary without the lemmas they depend on.)

H-Lemma 26

1. $\texttt{daG}\,\Phi_{da} \to \texttt{xaG}\,(\texttt{rm\_da2xa}\,\Phi_{da})$.
2. $\texttt{daG}\,\Phi_{da} \to \texttt{aG}\,(\texttt{rm\_da2a}\,\Phi_{da})$.

H-Corollary 27 (C-Strengthening/Weakening)

1. $\texttt{daG}\,\Phi_{da} \to \{\Phi_{da} \vdash_n \langle \texttt{is\_tm}\,T\rangle\} \to \{\texttt{rm\_da2xa}\,\Phi_{da} \vdash_n \langle \texttt{is\_tm}\,T\rangle\}$.
2. $\texttt{daG}\,\Phi_{da} \to \{\texttt{rm\_da2xa}\,\Phi_{da} \vdash_n \langle \texttt{aeq}\,T\,T'\rangle\} \to \{\Phi_{da} \vdash_n \langle \texttt{aeq}\,T\,T'\rangle\}$.
3. $\texttt{daG}\,\Phi_{da} \to \{\texttt{rm\_da2a}\,\Phi_{da} \vdash_n \langle \texttt{aeq}\,T\,T'\rangle\} \longleftrightarrow \{\Phi_{da} \vdash_n \langle \texttt{aeq}\,T\,T'\rangle\}$.

With the above lemmas, we can now promote H-Theorems 5 and 7.

H-Lemma 28 (Promotion)

1. $\texttt{daG}\,\Phi_{da} \to \{\Phi_{da} \vdash_n \langle \texttt{is\_tm}\,M\rangle\} \to \{\Phi_{da} \vdash_n \langle \texttt{aeq}\,M\,M\rangle\}$.
2. $\texttt{daG}\,\Phi_{da} \to \{\Phi_{da} \vdash_n \langle \texttt{aeq}\,M\,N\rangle\} \to \{\Phi_{da} \vdash_n \langle \texttt{aeq}\,N\,M\rangle\}$.
3. $\texttt{daG}\,\Phi_{da} \to \{\Phi_{da} \vdash_n \langle \texttt{aeq}\,M\,L\rangle\} \to \{\Phi_{da} \vdash_n \langle \texttt{aeq}\,L\,N\rangle\} \to \{\Phi_{da} \vdash_n \langle \texttt{aeq}\,M\,N\rangle\}$.

H-Theorem 29 (Completeness, G Version)

$\texttt{daG}\,\Phi_{da} \to \{\Phi_{da} \vdash_n \langle \texttt{deq}\,M\,N\rangle\} \to \{\Phi_{da} \vdash_n \langle \texttt{aeq}\,M\,N\rangle\}$.

Proof The steps of the `de_r` and `de_t` cases are the same as in the R version, using promotion, height weakening, and in the latter case also the induction hypothesis. The `de_l` case also uses height weakening, and in addition requires both *d-wk* and *d-str*.[15] □

6.3.3 Internal Adequacy

Although internal adequacy results were not needed to complete the above proofs, they are important, as already discussed. Here, we state the R version of adequacy for the `aeq` judgment.

[15] Also in this proof and a few others (e.g., H-Theorem 30 and H-Attempt 35), the inversion step uses specialized inversion lemmas, whose proofs follow from Coq's standard inversion.

H-Theorem 30 (Internal Adequacy of Algorithmic Equality)

$\texttt{xaR}\ \Phi_x\ \Phi_a \rightarrow \{\Phi_a \vdash_n \langle \texttt{aeq}\ M\ N\rangle\} \rightarrow \{\Phi_x \vdash_n \langle \texttt{is\_tm}\ M\rangle\} \wedge \{\Phi_x \vdash_n \langle \texttt{is\_tm}\ N\rangle\}.$

Although such results often require their own versions of helper lemmas such as membership and *c-str*, their proofs present no further complications. G versions of adequacy can also be stated and proved, again presenting no further complications. Similar internal adequacy statements have been proved for all other benchmarks formalized in Hybrid.

6.4 Order

As mentioned in Section 5, exchange is covered by SL-Theorem 1 (b), which expresses weakening, contraction, and exchange. Theorem 23 in Section 2.2, however, cannot be formalized in Hybrid, not because of any problem with handling exchange, but because specialized substitution lemmas of this particular form cannot be proved in Hybrid. The proof shares some similarities with the proof of type preservation for parallel reduction, which also cannot be formalized in the current version of Hybrid. We illustrate the problem in Section 6.6.

6.5 Uniqueness

For the Hybrid proof of type uniqueness (Theorem 24 in Section 2.2), we adapt the one from [15]. The definitions of the Hybrid constants used in this subsection include `app` and `lam`, where the latter now takes two arguments with the first representing the type of the bound variable. (See the `Syntax` section in Appendix A.3.) The definition of `atm` includes predicates `is_tm` and `oft`. The types are those of the simply-typed lambda-calculus, and thus there are no operators with binders. We take advantage of the underlying RL, and define types inductively in Coq, and redefine the `cLAM` constant and `lam` constructor to take into account the type of the bound variable.

$$\begin{array}{l}\texttt{Inductive tp} \quad := \texttt{i} : \texttt{tp} \mid \texttt{arr} : \texttt{tp} \rightarrow \texttt{tp} \rightarrow \texttt{tp}.\\ \texttt{Inductive con} \ := \texttt{cAPP} : \texttt{con} \mid \texttt{cLAM} : \texttt{tp} \rightarrow \texttt{con}.\\ \texttt{Definition lam} := \lambda M, A.\ \texttt{APP}\ (\texttt{CON}\ (\texttt{cLAM}\ A))\ (\texttt{LAM}\ \lambda x.M\ x).\end{array}$$

Uniqueness of variables in a context does not hold if we consider the specification of the SL alone. In particular, recall the specification of the `s_all` rule introduced on page 29. In this rule x represents an arbitrary term of type `expr`, and we cannot assume that it is a variable. For the special case of the type uniqueness benchmark, we showed in [15] that we can achieve uniqueness of context variables with only a surprisingly minimal amount of additional formal infrastructure. In particular, we used Hybrid's `VAR` constructor to encode free variables, and added a definition (`nvC`) that provided the capability of creating a new variable that is *fresh* with respect to a context. In particular (`VAR` v) represents a variable, where v is a natural number, and (`nvC` Φ) returns the number that is one greater than the maximum of all the numbers representing free variables in Φ. All that is required is a small library of simple lemmas about `nvC` (see [15]). With this extra infrastructure, we cannot use the definition of the `xtG` schema in Appendix A.3 as directly as in the other benchmarks. In particular, the main clause of the inductive definition of `xtG` is:

$$\texttt{xtG}\ \Phi_t \rightarrow \texttt{xtG}\ ((\texttt{oft}\ (\texttt{VAR}\ (\texttt{nvC}\ \Phi_t))\ A) :: \Phi_t)$$

Note that definition omits well-formedness annotations from the context as in several other benchmarks. We also need 3 membership lemmas. The first two rule out non-variables from the context, and the third expresses the required uniqueness constraints for the context.

H-Lemma 31 (**Context Membership**)

1. $\texttt{xtG}\ \Phi_t \rightarrow (\texttt{oft}\ (\texttt{lam}\ B\ M)\ A) \notin \Phi_t$
2. $\texttt{xtG}\ \Phi_t \rightarrow (\texttt{oft}\ (\texttt{app}\ M\ N)\ A) \notin \Phi_t$
3. $\texttt{xtG}\ \Phi_t \rightarrow (\texttt{oft}\ M\ A \in \Phi_t) \rightarrow (\texttt{oft}\ M\ B \in \Phi_t) \rightarrow \exists i.(M = (\texttt{VAR}\ i) \wedge A = B)$.

H-Theorem 32 (**Type Uniqueness**)

$\texttt{xtG}\ \Phi_t \rightarrow \{\Phi_t \vdash_n \langle \texttt{oft}\ M\ A\rangle\} \rightarrow \{\Phi_t \vdash_m \langle \texttt{oft}\ M\ B\rangle\} \rightarrow A = B$.

Proof By complete induction on n as usual. Inversion on $\{\Phi_t \vdash_n \langle \texttt{oft}\ M\ A\rangle\}$ breaks the proof into cases as usual. In the variable case, we know that context Φ_t contains $(\texttt{oft}\ M\ A)$, and by Lemma 31 (3) that M has the form $(\texttt{VAR}\ v)$. We can then apply inversion to the second sequent $\{\Phi_t \vdash_m \langle \texttt{oft}\ (\texttt{VAR}\ v)\ B\rangle\}$. Since there are no `prog` clauses for the variable case, by inversion we also conclude $(\texttt{oft}\ (\texttt{VAR}\ v)\ B) \in \Phi_t$, and thus by Lemma 31 (3), that $A = B$.

Case `of_a`: After several inversions on the first sequent, we get:

$$\frac{\texttt{h}_1 : \mathit{IH} \quad \texttt{h}_2 : \texttt{xtG}\ \Phi_t \quad \texttt{h}_3 : \{\Phi_x \vdash_{n-2} \langle \texttt{oft}\ M_1\ (\texttt{arr}\ A'\ A)\rangle\} \quad \texttt{h}_4 : \{\Phi_x \vdash_{n-2} \langle \texttt{oft}\ M_2\ A'\rangle\} \quad \texttt{h}_5 : \{\Phi_x \vdash_m \langle \texttt{oft}\ (\texttt{app}\ M_1\ M_2)\ B\rangle\}}{A = B}$$

At this point, inversion on $\texttt{h}_5$ results in two subcases. One is similar to the application cases of other proofs and can be completed by the induction hypothesis. The other introduces the new hypothesis $(\texttt{oft}\ (\texttt{app}\ M_1\ M_2)\ B) \in \Phi_t$, which we rule out by Lemma 31 (2). The `of_l` case makes a similar use of Lemma 31 (1). □

6.6 Substitution

The substitution lemma used in the type preservation for parallel reduction benchmark (Theorem 25 in Section 2.2) can be proven directly in Hybrid where parametric substitution is β-reduction and hypothetical substitution corresponds to an application of SL-Theorem 1 (c). Since we do not need uniqueness of variables, we redefine (the main clause of the definition of) context `xtG`:

$$\texttt{xtG}\ \Phi_t \rightarrow \texttt{xtG}\ ((\texttt{oft}\ x\ A) :: \Phi_t)$$

and prove a simple but necessary lemma:

H-Lemma 33 $\texttt{xtG}\ \Phi_t \rightarrow \{\Phi_t \vdash \langle \texttt{oft}\ M\ A\rangle\} \rightarrow \texttt{proper}\ M$.

H-Lemma 34 (**Substitution**)

$\texttt{xtG}\ \Phi_t \rightarrow (\forall x.\texttt{proper}\ x \rightarrow \{\texttt{oft}\ x\ A :: \Phi_t \vdash_n \langle \texttt{oft}\ (M\ x)\ B\rangle\}) \rightarrow \{\Phi_t \vdash_m \langle \texttt{oft}\ N\ A\rangle\} \rightarrow \{\Phi_t \vdash_{n+m} \langle \texttt{oft}\ (M\ N)\ B\rangle\}$.

The proof follows closely the informal proof of Lemma 25 in the companion paper.

As mentioned earlier, type preservation for parallel reduction as we have defined it here cannot be proved in the current version of Hybrid. This is an example where uniqueness of variables in the context is required, but the use of nvC as in the previous example is not enough. We illustrate the problem by showing one case where the proof gets stuck. As usual, a Coq inductive definition implements context relation (xrtR Φ_r Φ_t) from the Definitions section of Appendix A.4, omitting the well-formedness annotations for terms. Since proof heights are not important in illustrating the problem, we elide them except in the judgment that we induct over.

H-Attempt 35 (Type Preservation for Parallel Reduction)

xrtR Φ_r $\Phi_t \rightarrow \{\Phi_r \vdash_m \langle \texttt{pr1}\ M\ N\rangle\} \rightarrow \{\Phi_t \vdash \langle \texttt{oft}\ M\ A\rangle\} \rightarrow \{\Phi_t \vdash \langle \texttt{oft}\ N\ A\rangle\}$.

Proof We attempt to follow the informal proof of Theorem 26 in [17], here by using a complete induction on m, where we assume $i < m$ in the proof sketch below.

Case pr_l: We must show:

$$\frac{\begin{array}{l}\texttt{h}_1 : \mathit{IH} \quad \texttt{h}_2 : \texttt{xrtR}\ \Phi_r\ \Phi_t \quad \texttt{h}_3 : \{\Phi_r \vdash_i \langle \texttt{pr1}\ (\texttt{lam}\ M')\ (\texttt{lam}\ N')\rangle\} \\ \texttt{h}_4 : \{\Phi_t \vdash \langle \texttt{oft}\ (\texttt{lam}\ M')\ A\rangle\}\end{array}}{\{\Phi_t \vdash \langle \texttt{oft}\ (\texttt{lam}\ N')\ A\rangle\}}$$

Inversion on $\texttt{h}_4$ results in two subcases corresponding to the s_bc and s_init rules in Fig. 12 on page 26 as usual.

Subcase s_bc: This case is similar to the lam case of other proofs (such as the tm_l case of H-Theorem 2). We show some detail to help illustrate the problem. We first apply a few more inversion steps to $\texttt{h}_3$ and $\texttt{h}_4$. We also apply SL rules s_bc and s_all in a backward direction to the conclusion, obtaining the new subgoal:

$$\frac{\begin{array}{l}\mathit{IH} \quad \texttt{xrtR}\ \Phi_r\ \Phi_t \quad \texttt{h}_5 : \forall x.\texttt{proper}\ x \rightarrow \{\texttt{pr1}\ x\ x :: \Phi_r \vdash_{i-3} \langle \texttt{pr1}\ (M'\ x)\ (N'\ x)\rangle\} \\ \texttt{h}_6 : \forall x.\texttt{proper}\ x \rightarrow \{\texttt{oft}\ x\ A' :: \Phi_t \vdash \langle \texttt{oft}\ (M'\ x))\ B'\rangle\}\end{array}}{\forall x.\texttt{proper}\ x \rightarrow \{\Phi_t \vdash (\texttt{oft}\ x\ A')\ \texttt{imp}\ \langle \texttt{oft}\ (N'\ x)\ B'\rangle\}}$$

Applying Coq's ∀-introduction at this point introduces a new x, which we use to instantiate $\texttt{h}_5$ and $\texttt{h}_6$ and complete this case.

Subcase s_init: We have the following subgoal:

$$\frac{\begin{array}{l}\mathit{IH} \quad \texttt{xrtR}\ \Phi_r\ \Phi_t \quad \texttt{h}_5 : \{\Phi_r \vdash_i \langle \texttt{pr1}\ (\texttt{lam}\ M')\ (\texttt{lam}\ N')\rangle\} \\ \texttt{h}_6 : (\texttt{oft}\ (\texttt{lam}\ M')\ A) \in \Phi_t\end{array}}{\{\Phi_t \vdash \langle \texttt{oft}\ (\texttt{lam}\ N')\ A\rangle\}}$$

which is not provable. In order to prove it, we would need a lemma similar to H-Lemma 31, which requires restricting variables in contexts to be of the form (VAR i). Here, from such a lemma and $\texttt{h}_6$, we could derive a contradiction as desired. This kind of variable restriction would have to be built into the definition of xrtR. Then this subcase becomes provable, but the previous one can no longer be proved. This is because the variable x introduced by ∀-introduction in that proof is an arbitrary term of type expr. Since it doesn't necessarily have the form (VAR i), the context relation would not hold, and thus it would not be possible to apply the induction hypothesis. □

Fixing this problem is the subject of current work in Hybrid. One approach is to use a form of "weak" HOAS that replaces the s_all rule (see page 29) with one that explicitly

adds variables (using the VAR constructor) to the context. A better approach, which would not require sacrificing the benefits of full HOAS, is to implement the ∇-quantifier in Hybrid. Possible solutions are further discussed in Section 8.2.

7 Abella

Abella's RL ($\mathscr{G}$) is an intuitionistic first-order sequent calculus endowed with a fixed-point approach to (co)induction and with the nabla quantifier ∇ to support reasoning over λ-term syntax. We refer to [21, 22] for a detailed account of Abella and its meta-theory and stick to the bare minimum.

Differently from Hybrid, the Abella system implements the two-level approach in a way that limits syntactic overhead. The system accepts `prog` clauses in the form of (executable) λ-Prolog files [28], analogously to how Beluga interacts with LF signatures. For example, the concrete syntax for the specification of algorithmic equality is:

```
aeq (app M1 M2) (app N1 N2) :- aeq M1 N1, aeq M2 N2.
aeq (lam M) (lam N)         :- pi x\ aeq x x ⇒ aeq (M x) (N x).
```

Abella hides details such as the height of derivations by the displaying `seq` predicates with syntactic sugar and using an annotation-based treatment of induction. Abella also embeds meta-theoretic properties of the specification logic into *tactics* named `monotone`, `cut`, `inst`, which correspond respectively to weakening (including exchange and contraction), cut and specialization in SL-Theorem 1. Generic judgments are handled by the nabla quantifier, i.e., the SL rule `s_all` for universal quantification depicted in Fig. 12 is realized as:

$$\nabla x.\, \{\Gamma \vdash G\ x\} \rightarrow \{\Gamma \vdash \forall_\tau(\lambda x.\, G\ x)\}$$

As shown in [22], the nabla quantifier ensures freshness and correct scoping both at the left and right of the sequent, while recovering the usual instantiation properties of universal quantification at the SL level typical of the specialization rule.

As we have mentioned before, Abella 2.0 offers a n-ary SL, which generalizes the one described in Section 5 by replacing backchaining with *focusing*. In other words, Abella 2.0 now handles as `prog` clauses any hereditary Harrop formula (as before, without predicate quantification). More precisely we add to the goal reduction sequent $\Sigma; \Gamma \vdash G$ a left focusing judgment $\Sigma; \Gamma, [F] \longrightarrow_{\mathscr{C}} A$, where F is the formula under (left) focus; accordingly, we substitute the single left backchaining rule (bc) in Fig. 11 with the standard focused left rules for universal quantification, implication and conjunction, omitted here—see [47]. We only record the different shape of the axiom rule (match) and the one replacing backchaining (focus), as they have a different look w.r.t. proof scripts' variable cases and related member lemmas:

$$\frac{}{\Sigma; \Gamma, [A] \longrightarrow_{\mathscr{C}} A}\ \text{match} \qquad \frac{F \in \Gamma \cup \mathscr{C} \qquad \Sigma; \Gamma, [F] \longrightarrow_{\mathscr{C}} A}{\Sigma; \Gamma \longrightarrow_{\mathscr{C}} A}\ \text{focus}$$

Note that for the set of benchmarks that we consider in this paper, which are all second-order, the n-ary SL essentially collapses to the previous one.

7.1 Basic Linear Context Extensions

7.1.1 R Version

We encode our first context relation (xaR Φ_x Φ_{xa}) with an Abella inductive definition (keywords **Define** and :=) for the binary predicate xaR over the built-in sort of (atomic) assumptions **olist**:

```
Define xaR : olist → olist → prop by
         xaR  nil nil;
 nabla x, xaR (is_tm x :: Ts) (aeq x x :: As) := xaR Ts As.
```

Recall that the encoding of aeq and hence of Φ_{xa} does not contain the atom is_tm x, but the latter is needed to mimic induction over individuals, namely for the proof of A-Theorem 2 below. We start our development with a prototypical member lemma, namely the formalization of Lemma 6, where member is Abella's built-in predicate for "$\in$".

A-Lemma 1 (**Context Membership**)

```
member_is_tm: forall  Ts As F,
 xaR Ts As → member F Ts → exists  M, F = is_tm M ∧ member (aeq M M) As.
```

Proof By induction on the second premise. □

The proof of such lemmas is completely routine; unfortunately, the user has to state and reprove this kind of result every time, as no tactical language is available in Abella per se.[16] Note also the difference with the statement of the same result in Hybrid (H-Lemma 1): while in Hybrid's atomic contexts the predicate is immediately visible, in Abella 2.0 it needs to be singled out for focusing.

We are now ready for our first proof, which we will report in Abella's original notation (inspired by Coq). From the next proof on, we will somewhat streamline it. We will also omit outermost universal quantifiers, unless there is some interesting quantifier alternation.

A-Theorem 2 (**Admissibility of Reflexivity, R Version**)

```
xaR Ts As → {Ts ⊢ is_tm M} → {As ⊢ aeq M M}.
```

Proof By induction on the derivation of the second premise. The predicate on which we are carrying our induction over, here {Ts ⊢ is_tm M}*, is annotated with a star that is used to track the size of inductive arguments—we will see it in action in the tm_a case.

After fixing variables and assuming the antecedents, a call to the case tactic, which implements inversion, on {Ts ⊢ is_tm M} yields three cases:

Case "variable":

```
Variables: As, Ts, M
IH : forall  As Ts, xaR Ts As → {Ts ⊢ is_tm M}* → {As ⊢ aeq M M}
H1 : xaR Ts As
H2 : {Ts, [F] ⊢ is_tm M}*
H3 : member F Ts
------------------------
   {As ⊢ aeq M M}
```

[16] But see [5] for progress on this topic, as we touch upon in the conclusion.

Here we appeal to A-Lemma 1 (concrete syntax `apply member_is_tm to H1 H3.`):

```
Variables: As, Ts, M, M'
IH : forall  As Ts T, xaR Ts As → {Ts ⊢ is_tm T}* → {As ⊢ aeq M T}
H1 : xaR Ts As
H2 : {Ts, [is_tm M'] ⊢ is_tm M}*
H3 : member (is_tm M') Ts
H4 : member (aeq M' M') As
------------------------
   {As ⊢ aeq M M}
```

A further inversion on `H2` unifies `M` with `M'`, in particular in assumption `H4`. Then we use the `search` tactic, which coincides with Prolog search, to finish the proof. This implicitly uses first the focus and then the match rule in the SL.

Case `tm_a`:

```
Variables: As, Ts, M, N
IH : forall  As Ts M, xaR Ts As → {Ts ⊢ is_tm M}* → {As ⊢ aeq M M}
H1 : xaR Ts As
H3 : {Ts ⊢ is_tm N}*
H4 : {Ts ⊢ is_tm M}*
------------------------------------------------------------------------
   {As ⊢ aeq (app M N) (app M N)}
```

Although Abella's proofs by induction are carried out over the height of the derivation, the system does not show the numbers, as Hybrid does; rather it employs a form of abstract interpretation, whereby the annotation $(*)$ indicates that the height has indeed decreased and that the IH can be safely applied. Let us do that to H1 H3 (`apply IH to H1 H3`) and to H1 H4 and conclude the case by search, which in this case will entail selecting for focus (SL rule `prog`) clause `ae_a`.

Case `tm_l`:

```
Variables: As, Ts, T, M
IH : forall  As Ts T, xaR Ts As → {Ts ⊢ is_tm T}* → {As ⊢ aeq M T}
H1 : xaR Ts As
H3 : {Ts, is_tm x ⊢ is_tm (M x)}*
------------------------------------
   {As ⊢ aeq (lam M) (lam M)}
```

First, note that `x` is a fresh name, globally bound in the proof, generated by interpreting the object-level universal quantifier `pi` by ∇. Now, we would like to apply the inductive hypothesis as before. This is however allowed only if the contexts in `H1` and `H3` match, i.e., if `xaR (is_tm x :: Ts) (aeq x x :: As)` holds, see the appeal to rule $crel_{xa}$ in the proof of Theorem 7 in the companion paper. We can direct Abella to acknowledge this by the tactic `assert` or by using the underscore notation `apply IH to_ H3`, by which the system "guesses" the right context extension. Then we conclude by search. □

What about symmetry (the same holds for transitivity, which we omit here)? We state and prove it in the context predicate `aG`, which contains assumptions of the form `aeq x x`. We first need the encoding of the inversion Lemma 9.

A-Lemma 3 (**Context Inversion**)

```
member_aeq_inv: aG As → member F As → exists  M N, F = aeq M N ∧ M = N.
```

A-Theorem 4 (**Admissibility of Symmetry**)

```
aG As → {As ⊢ aeq M N} → {As ⊢ aeq N M}.
```

Proof By now standard induction on the second derivation, using A-Lemma 3 in the variable case. □

If we want to "package up" the fact that algorithmic equality is an equivalence relation, we need to lift the above result(s) to the "world" in which reflexivity is stated, namely the relation `xaR Ts As`. In this case this is easy, since we can prove by a trivial induction (similar to lemmas needed in Hybrid such as H-Lemma 22 (2)):

A-Lemma 5 `aG_xaR_proj: xaR Ts As → aG As`

This lemma just makes explicit the fact that if two contexts (instances) are related, then they satisfy their respective schema. This is internalized in Beluga's type theory, while it is a proof obligation in the much weaker-typed two-level approach.

Thanks to this lemma we can lift the relevant property to the larger world.

A-Corollary 6 (**Promotion of Symmetry**)

```
xaR Ts As → {As ⊢ aeq M N} → {As ⊢ aeq N M}.
```

7.1.2 G Version

We start by defining the generalized context

```
Define xaG : olist → prop by
 nabla x, xaG (is_tm x :: aeq x x :: Ga) := xaG Ga.
```

and prove the standard "member" lemma, which now has to take into account disjunctions of declarations:

```
A-Lemma 7 member_is_tm: xaG Ga → member E Ga →
    exists X, (E = is_tm X ∧ member (aeq X X) G) ∨ E = aeq X X.
```

Recall that the informal proof of reflexivity of Theorem 8 in the companion paper required a *d-strengthening* lemma, beyond *d-weakening*, stating that we can safely drop any `is_tm` assumption from a derivation of `aeq`, since the former judgment is irrelevant to the latter. See, for example, H-Corollary 4 in the Hybrid development, which follows from H-Lemma 3. Note that those results are stated for *any* context, without any associated schema. And therein lies the rub: in Abella 2.0 we do have to state theorems over open terms under a context schema or relation, so that the shape of the context can be delineated in the member lemma. However, if we state strengthening under `xaG Ga`, the statement will not be provable, as the context schema is not preserved in the proof. In particular, in the binder case, the context in the schema will not match the one passed to the `is_tm` judgment.

There is nevertheless a way out, as it has been pointed out by Todd Wilson (personal communication), whose solution we now follow. The trick is to define a "weaker" context predicate that will logically follow from `xaG`, while being flexible enough to match the

context passed in the binder case. Basically, we turn the declarations in the generalized context into *alternatives*:

```
Define xaG' : olist → prop by
  xaG' nil;
  nabla x, xaG' (is_tm x :: Ga x) := nabla x, xaG' (Ga x);
  nabla x, xaG' (aeq x x :: Ga x) := nabla x, xaG' (Ga x).
```

The fact that `xaG'` depends on `x` is crucial to make provable statements such as `xaG' (is_tm x :: aeq x x :: nil)` as well as `xaG' (is_tm x :: nil)`.

Armed with that we can state *d-str* in the weaker context predicate `xaG'`:

A-Lemma 8 (**D-Strengthening**)

```
forall Ga M N, nabla x, xaG' (Ga x) →
       {Ga x, is_tm x ⊢ aeq (M x) (N x)} → {Ga x ⊢ aeq (M x) (N x)}.
```

Proof Standard induction on the second derivation using for the variable case a member lemma identical to A-Lemma 7, but for `xaG'`. In the binder case the IH is applicable with the instantiation `Ga = x1\ aeq x2 x2 ::(Ga x1).` □

To use our strengthening, we will need to promote it to the "right" context and in fact we can immediately show by a straightforward induction

A-Lemma 9 `xaG_is_xaG': xaG Ga → xaG' Ga.`

We are now ready to prove:

A-Theorem 10 (**Admissibility of Reflexivity, G Version**)

```
xaG Ga → {Ga ⊢ is_tm M} → {Ga ⊢ aeq M M}.
```

Proof By induction on the second derivation. The only interesting case is the binder's:
Case $\mathtt{tm}_1$*:*

```
IH : forall Ga M, xaG Ga → {Ga ⊢ is_tm M}* → {Ga ⊢ aeq M M}
H1 : xaG Ga
H3 : {Ga, is_tm x1 ⊢ is_tm (M1 x1)}*
============================
 {Ga ⊢ aeq (lam M1) (lam M1)}
```

We help Abella to apply the IH by providing the correct instantiation by `apply IH to_ H3 with Ga = aeq x1 x1 :: is_tm x1 :: Ga`, yielding:

```
...
H1 : xaG Ga
H4 : {Ga, is_tm x1, aeq x1 x1 ⊢ aeq (M1 x1) (M1 x1)}
============================
 {Ga ⊢ aeq (lam M1) (lam M1)}
```

Now we "demote" `xaG` to `xaG'` with A-Lemma 9, apply D-strengthening and search. □

How well does this approach scale? We have carried out all the G versions of the benchmarks—they can be found in the ORBI repository with a `G` suffix—and we can antic-

ipate these further observations, which are better appreciated after reading through the R versions. In general, member lemmas are somewhat more verbose to state, due to the presence of disjunctions; their proofs are still standard. However, they need to be duplicated for the weaker context. The type preservation case study is completely analogous to the present one. The polymorphic lambda calculus benchmark requires four strengthening lemmas, in a specific order, and the member lemmas get more and more unwieldy. For completeness, we need to lift the equivalence relation property of `aeq` from the schema `aG` and `xaG` to `daG` (and similarly for pairwise substitution). Since there is no built-in support for context subsumption, that is *c-wk/str*, in Abella, one option is to do-it-ourselves with explicit strengthening functions, as in the Hybrid section; there is the additional complication that we have to encode these functions as *relations* and then prove that they are indeed total—functionality does not seem to play a role here. Another option is to just reprove them, as we did, in the "larger" context, meaning in the weakened version `daG'`, for them to be finally lifted to `daG` by means of a lemma analogous to A-Lemma 9.

For all these reasons, while we readily acknowledge that this may not be the final word on G-style proofs in Abella 2.0, we favor the R approach, as it yields more concise and modular proofs.

There is a more general and important lesson to learn: context predicates do not adequately encode schemas, since we have just exhibited a context predicate, namely `xaG'` validating run-time contexts that do not belong to the given schema, according to the rules of schema satisfaction given in Section 2.2 of the companion paper. We comment on this further in Section 8.2.

7.2 Linear Contexts Extensions with Alternative Declaration (R Version)

Here we depart from the methodological position we took in the companion paper and followed in the previous sections, where we stated theorems in the smallest context predicate (relation) possible—we called that "the fine-grained approach". We instead use here what could be seen as the *least upper bound* of the relevant schemas, here `atpG` and `aeqG`, in this case the latter. This makes the development much smoother, since it does not require promotion lemmas (Lemma 16), for which, as we have mentioned, Abella provides no intrinsic support. Note that the projection trick that we used before (A-Lemma 5) here does not apply. However, the "fine-grained" approach has been pursued in the Hybrid section and could be duplicated here, albeit more painfully. In any case, one can argue that eventually algorithmic type equality will have to be packaged up with term equality and we may as well formulate all statements in the smallest *common* context to begin with.

Having gone this way, the development is completely uneventful. Once we state the expected context relation:

```
Define aeqR : olist → olist → prop by
 nabla x, aeqR (is_tm x :: Ts) (aeq x x :: As) := aeqR Ts As;
 nabla a, aeqR (is_tp a :: Ts) (atp a a :: As) := aeqR Ts As.
```

it is just a matter of proving the two standard member lemmas and moving directly to our version, which subsumes Theorem 14, Lemmas 15 and 16, and finally Theorem 17:

A-Theorem 11 (**Admissibility of Reflexivity for Types and Terms**)

1. `aeqR Ts As → {Ts ⊢ is_tp A} → {As ⊢ atp A A}.`
2. `aeqR Ts As → {Ts ⊢ is_tm M} → {As ⊢ aeq M M}.`

Proof

1. By induction on the second derivation.
2. By induction on the second derivation, using point (1) in the tapp case.

□

7.3 Non-Linear Context Extensions (R Version)

The formalization of the completeness proof in Abella turns out to be slightly more challenging. As we have seen in Section 4.2 of the companion paper, the encoding of declarative equality "suffers" from the need to reflect at the judgmental level the sorting statement `M:tm`. Recall in fact that in the reflexivity case of the mathematical proof of Theorem 22 from the assumption $\Phi_{xd} \vdash \mathsf{deq}\ M\ M$ we were able to infer $\Phi_{xd} \vdash \mathsf{is\_tm}\ M$ by construction. In some cases, for example w.r.t. algorithmic equality, this invariant can be inductively established without further ado—note that this result is not needed in any of the following developments.

A-Theorem 12 (**Internal Adequacy of Algorithmic Equality**)

```
xaR Ts As → {As ⊢ aeq M1 M2} → {Ts ⊢ is_tm M1} ∧ {Ts ⊢  is_tm M2}.
```

Proof Usual induction on `{As ⊢ aeq M1 M2}` □

An analogous result holds for declarative equality only due to the fact that we have strategically added `is_tm` annotations, as discussed in Section 5, page 27. In these proofs two relations are at work: (`daR` Φ_{xa} Φ_{xd}) and (`xaR` Φ_x Φ_{xa}), the former in the main statement, the latter in the relational strengthening Lemma 20. It significantly simplifies the implementation if we join them together in a *ternary* relation, which works as a "bridge" between the two—see the Hybrid section for a strictly fine-grained version more closely connected to the analysis in Section 3.2.2 of the companion paper:

```
Define xadR : olist → olist → olist → prop by
 nabla x, xadR (is_tm x :: Ts) (aeq x x :: As) (deq x x :: is_tm x :: Ds)
       := xadR Ts As Ds.
```

Let us now consider the statement of our theorem

```
completeness: xadR Ts As Ds → {Ds ⊢ deq M1 M2} → {As ⊢ aeq M1 M2}.
```

First, we have to lift the equivalence properties of `aeq` to the current world `xadR`, easily accomplished analogously to what we have done in A-Lemma 5:

A-Lemma 13 `xaR_xadR_proj: xadR Ts As Ds → xaR Ts As`

A-Lemma 14 (**Promotion of Symmetry and Transitivity**)

```
xadR Ts As Ds → {As ⊢ aeq M N} → {As ⊢ aeq N M}.
xadR Ts As Ds →
       {As ⊢ aeq M1 M2} → {As ⊢ aeq M2 M3} → {As ⊢ aeq M1 M3}
```

This is not quite enough: consider the case de_r of the completeness proof:

```
xadR Ts As Ds    {Ds ⊢ is_tm M2}*
 ------------------------------
  {As ⊢ aeq M2 M2}
```

Here, in order to apply our reflexivity A-Theorem 2, we need to *c-strengthen* the assumption {Ds⊢ is_tm M2} to {Ts⊢ is_tm M2}. First we state a by now familiar member lemma:

A-Lemma 15 `member_deq: xadR Ts As Ds → member F Ds →`
`exists M, (F = is_tm M ∨ F = deq M M) ∧`
`member (aeq M M) As ∧ member (is_tm M) Ts.`

Although the statement is somewhat more involved, the proof is still elementary.

The proof of *c-strengthening* is actually not difficult thanks to the way Abella implements SL-weakening:

A-Lemma 16 `is_tm_cstr: xadR Ts As Ds → {Ds⊢ is_tm M} → {Ts⊢ is_tm M}.`

Proof By induction on the second derivation. The interesting case is the binder:

```
... H3 : {Ds, is_tm x ⊢ is_tm (M x)}*
---------------------------------------
 {Ts ⊢ is_tm (lam M)}
```

Here we need to weaken H3 so as to apply the IH. The key insight is *not* to do it manually with `assert {Ds, is_tm x, deq x x⊢ is_tm (M x)}`, because this would loose the * annotation and the IH would not be applicable anymore. Rather we directly use SL weakening with the tactic `monotone H3 with (deq x x :: is_tm x :: Ds)` yielding

```
... H4 : {Ds, is_tm x, deq x x ⊢ is_tm (M x)}*
  ---------------------------------------------
   {Ts ⊢ is_tm (lam M)}
```

Now the IH is applicable and the rest of the proof is routine. □

A-Corollary 17 (**Promotion of Reflexivity**)

`xadR Ts As Ds → {Ds ⊢ is_tm M} → {As ⊢ aeq M M}.`

Proof By A-Theorem 2 and A-Lemma 16. □

A-Theorem 18 (**Completeness**)

`xadR Ts As Ds → {Ds ⊢ deq M1 M2} → {As ⊢ aeq M1 M2}.`

Proof By induction on the second premise. We show a couple of cases:

Case de_r: By A-Corollary 17.

Case de_t:

```
xadR Ts As Ds    {Ds ⊢ deq M1 M'}*   {Ds ⊢ deq M' M2}*
  ------------------------------------------------------
   {As ⊢ aeq M1 M2}
```

Here we simply apply the IH twice and then appeal to our promoted version of transitivity, A-Lemma 14. Symmetry is analogous, while the lambda case is uneventful. □

7.4 Order (R Version)

This case study brings in some more sophisticated reasoning about variable identities; it is therefore convenient here and in the sequel to introduce the following definition, which recognizes when a term is in fact a (unique) name:

```
Define name : tm → prop by
 nabla x, name x.
```

Doing so this late in the game shows that the examples so far did not use in any essential way the "uniqueness" assumption in context schemas. But this is going to change from now on. Hence we restate A-Lemma 3 with the additional conclusion that `N` and hence `M` are indeed names:

A-Lemma 19 `member_aeq_inv_name: aG As → member F As →`
`exists M N, F = aeq M N ∧ M = N ∧ name N.`

The next statement uses nabla to bind the spot where substitution will occur.

A-Theorem 20 (**Pairwise Substitution**)

```
forall As M1 M2 N1 N2, nabla x, aG As →
         {As, aeq x x ⊢ aeq (M1 x) (M2 x)} → {As ⊢ aeq N1 N2} →
            {As ⊢ aeq (M1 N1) (M2 N2)}.
```

Proof By induction on the second derivation. Remarkably, the only interesting case is the variable one:

```
{As ⊢ aeq N1 N2}
H2: member (F x) (aeq x x :: As)
{As, aeq x x, [F x] ⊢ aeq (M1 x) (M2 x)}*
------------------------------------------------------
   {As ⊢ aeq (M1 N1) (M2 N2)}
```

Case analysis on `H2` yields two sub-cases: `aeq (M1 x) (M2 x) = (aeq x x)`, hence `M1`, `M2` are the identity functions and the goal reduces to {`As ⊢ aeq N1 N2`}. The second sub-case is when `member (F x) As`. By A-Lemma 19 we know `M1 = M2` and `(M2 n)` is a name. This entails two additional possibilities. First, it could be a different name `y`, leading to the goal {`As y⊢ aeq y y`}, which follows immediately since we have `member (aeq y y) (As y)`. Alternatively, as before, `M1` could be the identity function and again the goal is {`As ⊢ aeq N1 N2`}, which we have.

The binder case follows straightforwardly by induction, since the SL treats object level contexts modulo SL-weakening. □

Were we, for whatever reason (see [29, 31] for a particular perspicuous instance), to treat contexts explicitly at the meta-level, that is, had `aeq` the type **olist** → `tm` → `tm` → `prop`, we would have to formalize exchange as an inductive relation, in a "structural" enough way to work well with the two-level reasoning style; this is not completely trivial, as exhibited in the cited papers.

7.5 Uniqueness (R Version)

This benchmark is explained in full detail in [22], so we keep it short. The schema definition

```
Define xtG : olist → prop by
 nabla x, xtG (oft x A :: Os) := xtG Os.
```

says that `xtG Os` holds if and only if `Os` is a list of atoms `oft x A` in which `x` is a nominal constant that does not appear elsewhere in the list; thus its typing must be unique, a condition enforced by means of the correct alternation between nabla and the outermost universal quantification. Compare it to a nabla-less version:[17]

```
Define xtG' : olist → prop by
 xtG' (oft X A :: Os) :=
          (forall  M N, X = app M N → false  ) ∧
          (forall  A M, X = abs A M → false  ) ∧
          (forall  A', member (oft X A') Os → false  ) ∧  xtG' Os.
```

Here we must explicitly restrict structure and occurrences of `X`, as we have no way to internalize the fact that this is indeed a name.

Two member lemmas are required, one to prove the desired property for the variable case:

```
xtG_uq: xtG Os → member (oft M A) Os → member (oft M A) Os → A = B.
```

the other to rule out impossible cases when inverting on the typing relation:

```
member_of_name: xtG Os → member E Os → exists  M A, E = oft M A ∧ name M.
```

In fact, inversion on a judgment such as {`Os ⊢ oft (lam A1 M) A2`} not only yields, by backchaining, the expected premise {`Os, oft x A1 ⊢ oft (M x) A2`}, but also, by the SL match/init rule, the premise `member (oft (lam A1 M) N) Os`. However, the above lemma gives us the absurd hypothesis `name (lam A1 M)` and hence this case is immediately discharged.

A-Theorem 21 (**Type Uniqueness**)

```
xtG Os → {Os ⊢ oft M A1} → {Os ⊢ oft M A2} → A1 = A2.
```

Proof By induction on {`Os ⊢ oft M A1`} and inversion on {`Os ⊢ oft M A2`}, using `xtG_uq` in the variable case and `member_of_name` in the case analysis of {`Os ⊢ oft M A2`}. □

7.6 Substitution (R Version)

The context relation is unsurprising:

```
Define xrtR : olist → olist → prop by
 nabla x, xrtR (pr x x :: Ps) (oft x A :: Os) := xrtR Ps Os.
```

[17] Recall that Hybrid uses the `nvC` function for new variables, requiring H-Lemma 31.

We need two familiar member lemmas, the above `member_of_name` to clean up inversion plus a version of the inversion A-Lemma 3 for `pr` after which we can state the Abella version of Theorem 23:

A-Theorem 22 (**Type Preservation for Parallel Reduction**)

```
xrtR Ps Os → {Ps ⊢ pr M1 M2} → {Os ⊢ oft M1 A} → {Os ⊢ oft M2 A}.
```

Proof By induction on the derivation of {`Ps ⊢ pr M1 M2`} and inversion on the derivation of {`Os ⊢ oft M1 A`}. We show only two cases:

Case "variable":

```
  xrtR Ps Os   {Os ⊢ oft M1 A}  ... member F Ps
  ------------------------------------------------
   {Os ⊢ oft M2 A}
```

Here we use the inversion lemma to enforce that `M1 = M2`—note that we do not need the fact that they are indeed names in this case, although that is actually required to rule out impossible typing premises in the `pr_l` and `pr_a` cases, as explained in Section 7.5.

Case `pr_b`:

```
  {Os ⊢ oft (app (lam A1) N1) T}   {Ps, pr x x ⊢ pr (M1 x) (M2 x)}*
  {Ps ⊢ pr N1 N2}*
-------------------------------------------------------------------------
   {Os ⊢ oft (M2 N2) T}
```

The non trivial inversion on the first premise yields the two assumptions `H' : {Os ⊢ oft (lam M1) (arr B A)}` and {`Os ⊢ oft N1 B`}. By IH we have that {`Os ⊢ oft N2 B`}. We invert further on `H'` and get:

```
... H: {Ps, pr x x ⊢pr (M1 x) (M2 x)}*   {Os, oft x B ⊢ oft (M1 x) A}
-------------------------------------------------------------------------
   {Os ⊢ oft (M2 N2) A}
```

We apply the IH and have {`Os, oft x B⊢ oft (M2 x) A`}. The informal proof here applies the substitution lemma to get the desired conclusion. So we instantiate `n` in `H` with the appropriate term, namely `N2` (tactic `inst`). Here we see that the nabla quantifier not only provides the correct scoping mechanism but also the required substitution behavior.

```
  {Os ⊢ oft N2 B}          {Os, oft N2 B ⊢ oft (M2 N2) A}
-----------------------------------------------------------
   {Os ⊢ oft (M2 N2) A}
```

Now apply `cut` and it is all over. □

8 Summary and Discussion

Systems supporting HOAS fall into two categories. On one side of the spectrum, we have systems built on a type-theoretic foundation such as Twelf Beluga. On the other side there are systems such as Abella and Hybrid that are based on a proof-theoretic foundation and

follow the two-level approach, implementing a specification logic inside a logic or type theory.

8.1 Systems Based on Type Theory

Both Twelf and Beluga encode formal systems in the logical framework LF. In both systems *d-weakening* and *d-exchange* are built into the underlying type theory of LF, while *d-strengthening* is supported via subordination. Applying substitution lemmas (parametric and hypothetical) is simply β-reduction.

However, the two systems differ in the encoding of proofs. While Beluga provides a separate reasoning language where we are able to abstract over contexts of assumptions, in Twelf we remain within the logical framework LF and represent proofs as relations. This has several consequences. In Twelf, there is no simple way of inspecting a context and writing a generic variable case. All assumptions are implicit. Instead of a generic base case that applies to all variables in a given context, the proof for the base case is introduced whenever we encounter a variable or assumption. As a consequence, we are *assuming* that a given property that we want to prove holds for variables, instead of *proving* it for all variables. For example, we assume in the type uniqueness proof that every variable we introduce has a unique type instead of proving in general that whenever we introduce a variable to the context together with its type, it must be unique.

World checking then ensures that for all variables an appropriate base case has been introduced. Therefore, specifying worlds and checking that a given LF specification is consistent with it requires going outside of LF. In other words, while we write proofs as relations in LF, verifying that the relation constitutes a valid proof is accomplished by a variety of checkers (mode, coverage, termination, and world checkers) whose foundations lie outside of LF. Moreover, there is a conceptual gap between the on-paper proof, the relational representation of the proof in LF, and its interpretation as a total function, as established by the totality checker.

Proofs written as relations in Twelf live in an *ambient* context of assumptions and proving context membership lemmas such as Lemma 6 and Lemma 9 is unnecessary. The ambient context is the smallest general context containing all necessary hypothetical and parametric assumptions. World checking takes into account context subsumption and validates referring to a relation defined in a "smaller" world (for example `reflG`) while defining a relation in a "bigger" world (for example `ceqG`). Hence, promotion lemmas such as Lemma 12 and Lemma 18 are redundant. However, contexts are not first-class in Twelf; as such, we cannot abstractly reason about their properties. Twelf also lacks inductive definitions, and for good reasons; thus it cannot express context relations directly. As a consequence, proofs via context relations would require a substantial overhead of reifying assumptions explicitly (see for example [11]); it is unclear how to make the reasoning behind context subsumption (i.e., context strengthening combined with context weakening) explicit.

Beluga takes a different approach, providing a separate language for implementing proofs. The proof language supports abstractions over contexts of assumptions and schemas classifying contexts. Moreover, we encapsulate a LF object together with its context to form a contextual object. Unlike Twelf's global ambient context, every object in Beluga carries its own context in which it is meaningful. Beluga's reasoning language also supports splitting on contextual objects and contexts and even defining relations about them.

This has three consequences. First, we are able to write a generic base case by pattern matching on elements in the contexts and abstracting over objects denoting variables; in particular, coverage checking [12, 38] guarantees that given an object `[`γ $\vdash$ `M]` of type `[`γ $\vdash$

`tm]`, we know that either `M` is a variable from γ or an element constructed by the constants `lam` and `app`. Hence, proving context membership lemmas explicitly is unnecessary when proving the G version of properties.

Second, we inherit properties about abstract contexts such as uniqueness of declarations; we can also state properties about contexts using recursive types and move between contexts via substitutions; this gives the programmer considerable flexibility in how to implement proofs. Both the G version and the R version are possible, although the R version comes with a considerable overhead.

Third, there is no conceptual gap between the on-paper proof and its implementation as a function. In Beluga splitting an argument of a given type into different cases is done simply by looking at all possible constructors defining the given type and considering all possible assumptions that can be used to construct elements of this type. In fact, we can compile Beluga's case-expressions using standard pattern matching compilation strategies [18]. As such it is conceptually very similar to case analysis in Abella or Hybrid and much simpler than Twelf's coverage checking, which must guarantee both input and output coverage for a given relation and reason via worlds about the ambient context to ensure that all base cases are covered.

Moreover termination of a Beluga function can be verified directly during type checking. The situation in Twelf is again quite different. The termination checker in Twelf guarantees that the logic programming engine will terminate when running the relations [41]—it makes no direct claims about the total function (i.e., the proof) to which the relation corresponds. As Beluga allows us to directly encode the proof as a function, the flow of information in the proof corresponds directly to the sequence of steps in the program and establishing correctness of the mechanization is conceptually simpler and more direct.

Finally, we remark that world checking in Twelf establishes a property about the ambient context in which a given higher-order logic program is executed, while schema checking in Beluga guarantees that the contexts a given function manipulates are satisfying a given declared schema, i.e., they are well-typed. Contexts are first-class and Beluga's type theoretic foundation guarantees that we are manipulating well-formed contexts and well-formed derivations described using contextual objects and contexts. As in Twelf, Beluga's typing rules have *d-weakening* built-in and *d-strengthening* is supported in practice via subordination.

Similarly, Beluga supports context subsumption, i.e., it uses subordination to justify safe context weakening and strengthening. Hence, promotion lemmas such as Lemma 12 and Lemma 18 are redundant. To reduce the trusted base, one may wish to prove these lemmas explicitly. (See our earlier discussion in Section 4.7.1.) Beluga's language allows us to express context relations as indexed recursive datatypes (similar to inductive definitions) and hence the programmer can choose to make context strengthening and promotion explicit; this however significantly increases the amount of proofs one needs to write.

8.2 Systems Based on Proof Theory

Hybrid and Abella model hypothetical judgments of object logics using implication in the SL and parametric judgments via (generic) universal quantification. Hypothetical substitution is justified by appealing to the cut-admissibility lemma of the SL, i.e., via SL-Theorem 1 (c). Parametric substitution corresponds to the specialization rule of the corresponding meta-level universal quantifier, SL-Theorem 1 (d). Weakening is obtained directly from the specification logic, and so is exchange, via SL-Theorem 1 (b). Strengthening in these systems is proved on a case-by-case basis. There is so far no analogue to the

subordination dependency analysis implemented in Twelf and Beluga, although it is conceivable to do so. Note that all weakening and strengthening lemmas are applied explicitly via either an SL theorem or a specialized strengthening theorem. In Hybrid, some of these lemma applications can be automated via hints to the prover.

In both Abella and Hybrid contexts are represented inductively as lists of assumptions. The choice of this data structure is somewhat arbitrary and other notions such as (multi)sets have been used (Hybrid) or are scheduled to be used (Abella). Since these systems are untyped as far as contexts and schemas are concerned, the inductive specification of a context relation can play to a certain extent the role of the specification of a schema. More precisely, if a context belongs to a schema, then there is a simple Prolog-like proof in the RL that the corresponding context predicate holds for the list encoding that context. However, this encoding is not *complete*, since, as we have shown in Section 7.1.2, there are lists of declarations that happen to be provable for certain context predicates, but that do not satisfy the schema to which those predicates would map to. This is because the "cons" notation blurs the difference between the elements of a block (";") and sequences of blocks (","). If we want to enforce adequacy, in Abella we could refine the coding of contexts using λProlog's conjunction operator `&` to stand for ";". However, this is not fully supported in Abella yet, and using a do-it-yourself approach introducing say a constructor **block** : **olist** → o would significantly complicate context handling and go against Abella's and Hybrid's consolidated style.

With regard to the property of uniqueness of assumptions, Abella uses the ∇-quantifier to abstract over variables, unlike the standard universal quantifier that abstracts over terms. In Hybrid there is an unresolved tension between the loose "forall" interpretation of terms in contexts and the use of the `nvC` function explained in Section 6. In the former case, we encounter another source of incompleteness in the adequacy of schema satisfaction, since a Hybrid context (relation) definition will validate contexts with non-variable terms or with terms not occurring uniquely. With the latter, uniqueness holds, as seen in H-Lemma 31. However, we cannot always use this function, and even when we can, we lose the substitution property, see H-Attempt 35. In this regard, Abella's support for the ∇-quantifier is a real asset, as it provides both scoping and specialization in the SL. Implementing the ∇-quantifier in Hybrid (possibly following [4, 21]) is a subject of current work, which we expect will solve this problem.

In both Hybrid and Abella, splitting and induction on inductive predicates is supported as part of the proof-theoretical foundation, but splitting and inducting directly on terms is not. As mentioned in Section 5, one of the main reasons for introducing in Hybrid and Abella the `is_tm` predicate is to provide induction at the level of syntax.

Historically Hybrid was the first "mainstream" implementation of the two-level architecture [14, 30], as compared to the tool used in [27]. Hybrid can be seen as an "opened-up" Abella, explicit, easy to experiment with, and to trust. The system permits very flexible proof development; both the generalized and the relational approach can be accommodated, although typically the relational approach will lead to more compact proofs, since fewer infrastructural lemmas need to be established.

Another difference between these two systems is the way that induction is used. In Hybrid proofs are by *complete* induction on the explicit height of derivations in the SL; this is quite time consuming for the user, who needs to provide the correct instantiation of heights of sub-derivations. At worst, the correct height can be a non trivial function of the given derivations. For example, in Section 2.2 of [13], a `shape` judgment is defined that is weaker than `aeq`. In the electronic appendix to that paper, it was proved that, for a given a context relation on Φ_1 and Φ_2, if $\{\Phi_1 \vdash_n \langle \texttt{aeq}\ M\ N\rangle\}$ then $\{\Phi_2 \vdash_{2n+2} \langle \texttt{shape}\ M\ N\rangle\}$.

On the other hand, a proof by complete induction is just an application of Coq's `lt_wf_ind` theorem and therefore easy to check and trust. In Abella the induction tactic is a very clever piece of code that hides all the possibly complex details of height manipulation from the user, but at the same time is completely opaque. This may become problematic when we have to manipulate an assumption before applying the (co)inductive hypothesis; for example, if we apply a user-proven lemma, say strengthening, even one that preserves the proof height, there is no way to convince the system that applying the hypothesis afterward does not violate the inductive restrictions. In those cases we have to change the specification so as to introduce another OL judgment to induct on, one which is not affected by said lemma.

From a theoretic point of view, Gacek sketches in his dissertation [20] how to convert proofs following the annotation-based schema for induction into ones using $\mathscr{G}$'s original invariant-based induction rule. The consistency of the logic is shown by cut-elimination extending to $\mathscr{G}$ the cut-elimination proof, which was eventually published in [45].

Abella's RL is quite weak, in the De Bruijn sense [6], while Coq is among the strongest logical frameworks in existence. Beyond the obvious difference between simple and dependent types, which Hybrid does not really use that much, Hybrid inherits Coq's full recursive function space, and that is a mixed blessing. It allows us to define functions (as opposed to the relational-only approach of Abella), which is quite handy, as exemplified with respect to promotion lemmas; at the same time, it brings back all the (external) adequacy issues that have been studied in the last twenty years; hence the `abstr` annotations as explained in [16]. A related issue is the different equality relations of the two systems. In Abella, it is essentially $\alpha\beta\eta$-conversion, while Hybrid again inherits Coq's more complicated notion. This complication emerges, for example, in proofs by inversion, which in Hybrid sometimes require appeal to special equality and inversion lemmas.

Abella 2.0 is clearly more powerful than Hybrid as far as the SL is concerned, because it implements a n-ary logic instead of a second-order one, although the former is not conservative w.r.t the latter. It would be relatively easy to extend Hybrid's SL to the same one used by Abella; in fact, this could be done conservatively, as there would be no problem in having both SLs coexist, for the user to choose according to the nature of the benchmark. However, we feel, and we realize this is a judgment call, that n-ary benchmarks do not occur so frequently in the "wild" to make this change a priority. Besides, most of the time those benchmarks can be brought back in a standard way to the second-order case with a moderate increase in proof complexity. For an example, compare the third-order version of the `path` case study in [46] and its second-order rendering in [22].

There is also an additional Hybrid-specific issue: Hybrid's abstraction operator `LAM` is intrinsically second-order, and so is the `abstr` annotation. While a theory of n-ary abstractions is feasible, it has not been developed yet. Therefore, even with a n-ary SL, there would be some benchmarks, e.g., extending algorithmic equality to a third-order construct such as `callcc`, that Hybrid cannot currently handle.

8.3 Further Discussion and Summary

Based on the proof developments and the discussion so far, we can see that both Beluga and Abella provide support beyond what we typically achieve with HOAS. By extending LF with first-class contexts and contextual types, Beluga provides intrinsic support for abstracting over variables and contexts, and relating them via first-class substitutions and inductive definitions. The proof theory underlying Beluga is simple and requires only first-order logic with induction principles over a specific domain (i.e., contextual LF) and establishing normalization follows standard well-understood proof methods [38]. In Abella, we extend

first-order logic with iterative inductive definitions and the ∇-quantifier to abstract over objects denoting variables, but not contexts and, unless they carry no information such as in the case study reported by [1], contexts still need to be handled explicitly.

Although Hybrid's lack of intrinsic support for abstracting over variables increases the burden on the users compared to the other systems, it is the system with the richest type theory, since it is implemented as a Coq library. As a consequence, we can immediately take advantage of writing recursive functions in Coq instead of relations, avoiding infrastructural lemmas about determinacy and totality of relations, as would be needed in Abella. We cannot yet, however, write functions over Hybrid terms (`expr`). Hybrid also can readily take advantage of Coq's tactics, libraries, and decision procedures. Although the benchmarks we have considered in this paper do not showcase the need for libraries, they are of utmost importance in any real-life specifications. We however also acknowledge that choosing Coq as a system for implementing the Hybrid library means users work within a very rich type theory that, for example, also supports impredicative definitions, an extension often found to be controversial.

All systems but Twelf can accommodate both the generalized and the relational approach. In Hybrid proofs following the relational approach will be more compact, since fewer infrastructural lemmas need to be established. A similar remark applies to Abella, although *c-str* is handled differently. In Hybrid, the use of recursive functions to encode context strengthening leads to more compact proofs for both the G and R versions. In Beluga, proofs following the generalized approach will be even more compact due to the support for weakening and context subsumption provided by the underlying type theory.

Hybrid is arguably the most trustworthy system with respect to the form of proof certificates that it constructs. Coq builds proofs in the form of CIC lambda-terms, and these terms include every primitive inference, which in the case of Hybrid includes splitting, inductive reasoning, including reasoning about proof heights, and all applications of context lemmas, including weakening, strengthening, membership, etc. This Coq term can be independently checked by Coq's type checker, which simply verifies the correctness of each primitive inference. No extra external checkers are required. Note that this notion of trust concerns tool support for checking proofs, and does not include foundational strength and expressiveness, which is discussed above.

A Beluga proof is a program that can be checked independently by a small type checker, guaranteeing that contexts and contextual objects are well-formed. Beluga's type checker also verifies that all cases are covered exploiting subordination, while termination is validated by the programmer. Compared to Hybrid, Beluga programs are not compiled to a low-level core language in which all context reasoning including weakening and strengthening is explicit, nested pattern matches are translated into a series of single splits, and the termination measure is explicit. Although various steps have been taken into this direction—for example [18] describes how to compile pattern matching on contextual objects and [38] have recently proposed a core calculus for primitive recursive functions and proven that it is normalizing—presently Beluga programs are not compiled to this core foundation.

Twelf provides a compact type checker to verify that relations manipulate well-formed derivations taking into account weakening. However, to validate that the relation describes a total function, we rely on mode, world and totality checkers, which are outside of the LF framework and no direct certification exists.

Abella does not fit the LCF-style of proof assistants, nor does it provide a notion of proof certificate at this point; moreover, the annotation-based approach to (co)induction, as convenient as it is, introduces another level of indirection w.r.t. possible external checkers.

Table 1 At-a-glance features

	Twelf	Beluga	Abella 2.0	Hybrid
OL ctx	implicit	implicit	implicit	implicit
RL ctx	implicit	explicit	list of formulas	list of atoms
schemas	block decl	Σ-type	inductive predicate	inductive type
sch. sat	world checking	checking $\Phi : S_x$	proof of $(xC\ \Phi)$‡	proof of $(xC\ \Phi)$‡
d-wk	LF weakening	TT weakening	SL weakening	SL weakening
d-str	subordination†	subordination‡	case by case	case by case
c-wk/str	subsumption†	subsumption†	case by case	case by case
d/c-exc	LF exc †	TT exc†	admissible in SL	admissible in SL
uniq	Assumed†	Parameter var.	nabla in the head	nvC function†
p subst	β-red	β-red	spec of ∇ in SL	spec of $\forall$ in SL
h subst	β-red	β-red	cut in SL	atomic cut in SL†
G version	✓	✓	✓	✓
R version	×	✓	✓	✓

†: may not always work, ‡: sound but not complete

We end this section summing up some of the above discussion points in Table 1. The rows of the table summarize the following properties:

- The first two rows consider the representation of contexts both in OL specifications (OL ctx) and in the reasoning logic (RL ctx).
- The next two rows indicate how schemas are encoded (schemas) and how schema satisfaction is provided (schema sat).
- The following 4 rows indicate how we enforce structural properties (wk, str, exc) of contexts at the level of declarations (d-) and of whole contexts (c-).
- The next three row describes how we ensure uniqueness of the elements in a context (uniq), and the substitution properties, both parametric (p subst) and hypothetical (h subst).
- Finally, the last two rows state the feasibility of the G and R versions.

In the table, LF stands for "Logical Framework" and TT stands for "Contextual Modal Type Theory," the foundations of Twelf and Beluga, respectively. In both of these theories, Σ-types tie different declarations together.

9 Conclusion and Future Work

In this paper, we have formalized the benchmarks of the companion paper [17] in four systems: Twelf, Beluga, Hybrid, and Abella, and we have compared their general approach and solutions in some detail. In general, carrying out a qualitative comparison of a collection of very different systems is a daunting task, and this work represents a first step in this direction. This step involved focusing on one aspect, reasoning about contexts, and finding enough common ground to be able to present proofs of the benchmarks in a way that highlights some of the similarities and differences of the various systems. This common ground included first identifying the difference between the R and G versions, whose initial idea

can be seen in our early work [13], as well as developing a theory of contexts of assumptions, identifying the detailed structure of contexts along with their associated structural properties. This paper and the companion paper are very much intertwined. For example, our initial attempts at formalizing these benchmarks and comparing their proofs influenced the design of the theory of contexts, and vice versa, and the present paper is the result of several such cycles of evolution.

As mentioned, our initial benchmarks focus on contexts, and thus expanding this set of benchmarks is an important direction for future work. A variety of ideas in this direction are discussed in the conclusion of the companion paper, and so we do not repeat it further here. Another direction of future work involves formalizing these benchmarks in other systems, both those that support HOAS, as well as those that take other approaches. We hope that others will implement these benchmarks and further our understanding of the similarities and differences, strengths and weaknesses of different systems and approaches. The future evolution of the ORBI specification language to better support this task is also discussed in the companion paper.

In addition to providing a means for comparison, benchmark problems also *prescribe* a kind of standard to which systems should adhere. In continuing the development of the systems considered here (as well as other systems that adopt these benchmarks), it will be important to ensure that proofs of these problems can still be done; if they cannot or if the proofs become more difficult, it will be important to explain and understand why.

With regard to the systems discussed in this paper, our survey has already sparked a number of developments to address the shortcomings in the various approaches and systems. For example, implementing the ∇-quantifier is a current priority for Hybrid, and initial work on implementing a n-ary specification logic is underway. One of Abella's development branch offers a plug-in [5] that extends the system with a mechanism for declaring schemas and context relations in a way reminiscent of Beluga; the plug-in automatically derives standard context results, such as member lemmas stating inversion and uniqueness properties. Thanks to that, member lemmas can be in-lined and solved by calls to the plug-in tacticals, in many cases significantly simplifying proof scripts.[18]

Beluga is a system under active development and will continue to evolve. In the short term, we plan to support first-class substitutions also in the main release and improve upon source level syntax and reconstruction. Building on [38], we plan to support interactive development of proofs in Beluga to facilitate writing proofs. This is an important step to improve and ease writing Beluga code. The recent support of first-class substitutions and context relations via indexed recursive datatypes and their use in mechanizing the completeness proof of algorithmic equality discussed in this paper also provides new insights into how proofs in proof-theoretic systems relate to Beluga programs; we see this as a first step towards not only comparing systems by examples, but also for translating proofs done in one system to proofs in another.

Acknowledgments The first and third author acknowledge the support of the Natural Sciences and Engineering Research Council of Canada. We thank Kaustuv Chaudhuri, Andrew Gacek, Nada Habli, Dale Miller, Gopalan Nadathur and Yuting Wang for discussing some aspects of the work with us. A special thanks to Todd Wilson for suggesting how to do G proofs in Abella. The first author would also like to extend her

[18]There are currently some limitations that make the plug-in non applicable to benchmarks such as completeness of algorithmic equality, See [5] for more details and the Abella section of the ORBI repository for solutions for the doable ones.

gratitude to the University of Ottawa's Women's Writers Retreats. Finally, we thank the reviewers for their contrubutions in improving the paper.

Appendix A: ORBI Specifications of Challenge Problems

We give here the `Syntax`, `Judgments`, `Rules`, `Schemas`, and `Definitions` sections of the ORBI specifications for all the benchmarks presented in [17] and formalized in the main paper. The full ORBI files can be found at https://github.com/pientka/ORBI, and are called `EqualUntyped.orbi`, `EqualPoly.orbi`, `TypingSimple.orbi`, and `ParRed.orbi`, respectively. We include some context relations here (e.g., `alphaxR` and `aeqatpR`) that are not in the ORBI files because this paper discusses lemmas common to Hybrid and Abella R versions of proofs. Users carrying out these benchmarks in other systems might want the flexibility to structure proofs differently and possibly use different lemmas.

A.1 Algorithmic and Declarative Equality for the Untyped Lambda-Calculus

```
%% Syntax
tm: type.
app: tm -> tm -> tm.
lam: (tm -> tm) -> tm.

%% Judgments
aeq: tm -> tm -> type.
deq: tm -> tm -> type.

%% Rules
ae_a: aeq M1 N1 -> aeq M2 N2 -> aeq (app M1 M2) (app N1 N2).
ae_l: ({x:tm} aeq x x -> aeq (M x) (N x))
        -> aeq (lam (\x. M x)) (lam (\x. N x)).

de_a: deq M1 N1 -> deq M2 N2 -> deq (app M1 M2) (app N1 N2).
de_l: ({x:tm} deq x x -> deq (M x) (N x))
        -> deq (lam (\x. M x)) (lam (\x. N x)).
de_r: deq M M.
de_s: deq M1 M2 -> deq M2 M1.
de_t: deq M1 M2 -> deq M2 M3 -> deq M1 M3.

%% Schemas
schema xG:  block (x:tm).
schema xaG: block (x:tm; u:aeq x x).
schema xdG: block (x:tm; u:deq x x).
schema daG: block (x:tm; u:deq x x; v:aeq x x).

%% Definitions
inductive xaR: {G:xG} {H:xaG} prop =
| xa_nil: xaR nil nil
| xa_cons: xaR G H -> xaR (G, block x:tm) (H, block x:tm; u:aeq x x).

inductive daR: {G:xaG} {H:xdG} prop =
| da_nil: daR nil nil
| da_cons: daR G H -> daR (G, block x:tm; v:aeq x x)
                          (H, block x:tm; u:deq x x).
```

A.2 Algorithmic Equality for the Polymorphic Lambda Calculus

```
%% Syntax
tp: type.
arr: tp -> tp -> tp.
all: (tp -> tp) -> tp.

tm: type.


app: tm -> tm -> tm.
lam: (tm -> tm) -> tm.
tapp: tm -> tp -> tm.
tlam: (tp -> tm) -> tm.

%% Judgments
atp: tp -> tp -> type.
aeq: tm -> tm -> type.

%% Rules
at_al: ({a:tp} atp a a -> atp (T a) (S a))
        -> atp (all (\a. T a) (all (\a. S a)).
at_a: atp T1 T2 -> atp S1 S2 -> atp (arr T1 S1) (arr T2 S2).
ae_l: ({x:tm} aeq x x -> aeq (M x) (N x))
        -> aeq (lam (\x. M x)) (lam (\x. N x)).
ae_a: aeq M1 N1 -> aeq M2 N2 -> aeq (app M1 M2) (app N1 N2).
ae_tl: ({a:tp} atp a a -> aeq (M a) (N a))
        -> aeq (tlam (\a. M a)) (tlam (\a. N a)).
ae_ta: aeq M N -> atp T S -> aeq (tapp M T) (tapp N S).

%% Schemas
schema aG:   block (a:tp).
schema axG:  block (a:tp) + block (x:tm).
schema atpG: block (a:tp; u:atp a a).
schema aeqG: block (a:tp; u:atp a a) + block (x:tm; v:aeq x x).

%% Definitions
inductive alphxR: {G:aG} {H:axG} prop =
| ax_nil: alphxR nil nil
| ax_cons1: alphxR G H -> alphxR (G, block a:tp) (H, block a:tp).
| ax_cons2: alphxR G H -> alphxR  G              (H, block x:tm).

inductive atpR: {G:aG} {H:atpG} prop =
| atp_nil: atpR nil nil
| atp_cons: atpR G H -> atpR (G, block a:tp) (H, block a:tp; u:atp a a).

inductive aeqR: {G:axG} {H:aeqG} prop =
| aeq_nil: aeqR nil nil
| aeq_cons1: aeqR G H -> aeqR (G, block a:tp) (H, block a:tp; u:atp a a)
| aeq_cons2: aeqR G H -> aeqR (G, block x:tm) (H, block x:tm; v:aeq x x).

inductive aeqatpR: {G:atpG} {H:aeqG} prop =
| aa_nil: aeqatpR nil nil
| aa_cons1: aeqatpR G H ->
               aeqatpR (G, block a:tp; u:atp a a) (H, block a:tp; u:atp a a)
| aa_cons2: aeqatpR G H ->
               aeqatpR  G                         (H, block x:tm; v:aeq x x).
```

A.3 Static Semantics of the Simply-Typed Lambda-Calculus

```
%% Syntax
tp: type.
i: tp.
arr: tp -> tp -> tp.

tm: type.
app: tm -> tm -> tm.
lam: tp ->(tm -> tm) -> tm.

%% Judgments
oft: tm -> tp -> type.

%% Rules
oft_l: ({x:tm} oft x A -> oft (M x) B) ->
       oft (lam A (\x. M x)) (arr A B).
oft_a: oft M (arr A B) -> oft N A -> oft (app M N) B.

%% Schemas
schema xtG: block (x:tp; u:oft x A).
```

A.4 Parallel Reduction for the Simply-Typed Lambda-Calculus

```
%% Syntax
tp: type.
i: tp.
arr: tp -> tp -> tp.

tm: type.
app: tm -> tm -> tm.
lam: (tm -> tm) -> tm.

%% Judgments
oft: tm -> tp -> type.
pr: tm -> tm -> type.

%% Rules
oft_l: ({x:tm} oft x A -> oft (M x) B)
       -> oft (lam (\x. M x)) (arr A B).
oft_a: oft M1 (arr A2 A) -> oft M2 A2 -> oft (app M1 M2) A.

pr_l: ({x:tm} pr x x -> pr (M1 x) (M2 x))
       -> pr (lam (\x. M1 x)) (lam (\x. M2 x)).
pr_b: ({x:tm} pr x x -> pr (M1 x) (M2 x)) ->
          pr N1 N2 -> pr (app (lam (\x. M1 x)) N1) (M2 N2).
pr_a: pr M1 M2 -> pr N1 N2 -> pr (app M1 N1) (app M2 N2).

%% Schemas
schema xtG:  block (x:tm; v:oft x T).
schema xrG:  block (x:tm; u:pr x x).
schema xrtG: block (x:tm; u:pr x x; v:oft x T).

%% Definitions
inductive xrtR: {G:xrG} {H:xtG} prop =
| xrt_nil: xrtR nil nil
| xrt_cons: xrtR G H ->
            xrtR (G, block x:tm; u:pr x x) (H, block x:tm; v:oft x A).
```

References

1. Accattoli, B.: Proof pearl: Abella formalization of λ-calculus cube property. In: Second International Conference on Certified Programs and Proofs, Springer, LNCS, vol. 7679, pp. 173–187 (2012)
2. Ambler, S.J., Crole, R.L., Momigliano, A.: A definitional approach to primitive recursion over higher order abstract syntax. In: ACM Workshop on MEchanized Reasoning about Languages with varIable biNding, ACM Press, pp. 1–11 (2003)
3. Appel, A.W.: Verified software toolchain. In: Programming Languages and Systems, Springer, LNCS, vol. 6602, pp. 1–17 (2011)
4. Baelde, D.: On the expressivity of minimal generic quantification. In: Third International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2008, Elsevier, ENTCS, vol. 228, pp. 3–19 (2009)
5. Bélanger, O.S., Chaudhuri, K.: Automatically deriving schematic theorems for dynamic contexts. In: Ninth International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, ACM Press, International Conference Proceedings Series, pp. 9:1–9:8 (2014)
6. de Bruijn, N.G.: A plea for weaker frameworks. In: Huet, G., Plotkin, G. (eds.), pp. 40–67. Cambridge University Press, Logical Frameworks (1991)
7. Capretta, V., Felty, A.P.: Combining de Bruijn indices and higher-order abstract syntax in Coq. In: Types for Proofs and Programs, International Workshop, TYPES 2006, Springer, LNCS, vol. 4502, pp. 63–77 (2007)
8. Cave, A., Pientka, B.: Programming with binders and indexed data-types. In: Thirty-Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, pp. 413–424 (2012)
9. Cave, A., Pientka, B.: First-class substitutions in contextual type theory. In: Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, ACM Press, pp. 15–24 (2013)
10. Cave, A., Pientka, B.: Mechanizing logical relation proofs using contextual types theory. Tech. rep., School of Computer Science, McGill University (2014)
11. Crary, K.: Explicit contexts in LF (extended abstract). In: Third International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2008, Elsevier, ENTCS, vol. 228, pp. 53–68 (2009)
12. Dunfield, J., Pientka, B.: Case analysis of higher-order data. In: Third International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2008, Elsevier, ENTCS, vol. 228, pp. 69–84 (2009)
13. Felty, A., Pientka, B.: Reasoning with higher-order abstract syntax and contexts: A comparison. In: First International Conference on Interactive Theorem Proving, Springer, LNCS, vol. 6172, pp. 227–242 (2010)
14. Felty, A.P.: Two-level meta-reasoning in Coq. In: Fifteenth International Conference on Theorem Proving in Higher-Order Logics, Springer, LNCS, vol. 2410, pp. 198–213 (2002)
15. Felty, A.P., Momigliano, A.: Reasoning with hypothetical judgments and open terms in Hybrid. In: Eleventh ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming, ACM Press, pp. 83–92 (2009)
16. Felty, A.P., Momigliano, A.: Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. J. Autom. Reason. **48**(1), 43–105 (2012)
17. Felty, A.P., Momigliano, A., Pientka, B.: The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 1—a common infrastructure for benchmarks. CoRR (2015). arXiv:1503.06095
18. Ferreira, F., Monnier, S., Pientka, B.: Compiling contextual objects: Bringing higher-order abstract syntax to programmers. In: Seventh ACM SIGPLAN Workshop on Programming Languages Meets Program Verification, ACM Press, pp. 13–24 (2013)
19. Gacek, A.: The Abella interactive theorem prover (system description), vol. 5195, pp. 154–161 (2008)
20. Gacek, A.: A framework for specifying, prototyping, and reasoning about computational systems. PhD thesis, University of Minnesota (2009)
21. Gacek, A., Miller, D., Nadathur, G.: Nominal abstraction. Inf. Comput. **209**(1), 48–73 (2011)
22. Gacek, A., Miller, D., Nadathur, G.: A two-level logic approach to reasoning about computations. J. Autom. Reason. **49**(2), 241–273 (2012)
23. Habli, N., Felty, A.P.: Translating higher-order specifications to Coq libraries supporting Hybrid proofs. In: Third International Workshop on Proof Exchange for Theorem Proving, EasyChair Proceedings in Computing, vol. 14, pp. 67–76 (2013)

24. Harper, R., Licata, D.R.: Mechanizing metatheory in a logical framework. J. Funct. Program. **17**(4-5), 613–673 (2007)
25. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. J. Assoc. Comput. Mach. **40**(1), 143–184 (1993)
26. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
27. McDowell, R.C., Miller, D.A.: Reasoning with higher-order abstract syntax in a logical framework. ACM Trans. Comput. Log. **3**(1), 80–136 (2002)
28. Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge University Press (2012)
29. Momigliano, A.: A supposedly fun thing I may have to do again: A HOAS encoding of Howe's method. In: Seventh ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages, Theory and Practice, ACM Press, pp. 33–42 (2012)
30. Momigliano, A., Ambler, S.J.: Multi-level meta-reasoning with higher order abstract syntax. In: Sixth International Conference on Foundations of Software Science and Computational Structures, Springer, LNCS, vol. 2620, pp. 375–391 (2003)
31. Momigliano, A., Ambler, S., Crole, R.L.: A Hybrid encoding of Howe's method for establishing congruence of bisimilarity. Electr. Notes Theor. Comput. Sci. **70**(2), 60–75 (2002)
32. Momigliano, A., Martin, A.J., Felty, A.P.: Two-level Hybrid: A system for reasoning using higher-order abstract syntax. In: Second International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2007, Elsevier, ENTCS, vol. 196, pp. 85–93 (2008)
33. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Trans. Comput. Log. **9**(3), 1–49 (2008)
34. Pfenning, F.: Computation and deduction, http://www.cs.cmu.edu/~fp/courses/comp-ded/handouts/cd.pdf, accessed 14 October 2014 (2001)
35. Pientka, B.: Verifying termination and reduction properties about higher-order logic programs. J. Autom. Reason. **34**(2), 179–207 (2005)
36. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: Thirty-Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, pp. 371–382 (2008)
37. Pientka, B.: Programming inductive proofs: A new approach based on contextual types. In: Verification, Induction, Termination Analysis: Festschrift for Christoph Walther, Springer, LNCS, vol. 6463, pp. 1–16 (2010)
38. Pientka, B., Abel, A.: Structural recursion over contextual objects. In: Thirteenth International Conference on Typed Lambda Calculi and Applications, Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl (forthcoming) (2015)
39. Pientka, B., Dunfield, J.: Programming with proofs and explicit contexts. In: Tenth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM Press, pp. 163–173 (2008)
40. Pientka, B., Dunfield, J.: Beluga: A framework for programming and reasoning with deductive systems (system description). In: Fifth International Joint Conference on Automated Reasoning, Springer, LNCS, vol. 6173, pp. 15–21 (2010)
41. Rohwedder, E., Pfenning, F.: Mode and termination checking for higher-order logic programs. In: Programming Languages and Systems: Sixth European Symposium on Programming, Springer, LNCS, vol. 1058, pp. 296–310 (1996)
42. Schürmann, C.: The Twelf proof assistant. In: Twenty-Second International Conference on Theorem Proving in Higher Order Logics, Springer, LNCS, vol. 5674, pp. 79–83 (2009)
43. Schürmann, C., Pfenning, F.: Automated theorem proving in a simple meta-logic for LF. In: Fifteenth International Conference on Automated Deduction, Springer, LNCS, vol. 1421, pp. 286–300 (1998)
44. Schürmann, C., Pfenning, F.: A coverage checking algorithm for LF. In: Sixteenth International Conference on Theorem Proving in Higher Order Logics, Springer, LNCS, vol. 2758, pp. 120–135 (2003)
45. Tiu, A., Momigliano, A.: Cut elimination for a logic with induction and co-induction. J. Appl. Log. **10**(4), 330–367 (2012)
46. Wang, Y., Nadathur, G.: Towards extracting explicit proofs from totality checking in Twelf. In: Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, ACM Press, pp. 55–66 (2013)
47. Wang, Y., Chaudhuri, K., Gacek, A., Nadathur, G.: Reasoning about higher-order relational specifications. In: Fifteenth International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, ACM Press, pp. 157–168 (2013)
48. Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. In: Thirty-Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, pp. 427–440 (2012)